Aleksi Oinas

Kiinteistöhuollon tietojärjestelmän kehittäminen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Sähkötekniikan koulutusohjelma

Insinöörityö

11.5.2014

Metropolia

Tekijä Otsikko Sivumäärä Aika	Aleksi Oinas Kiinteistöhuollon tietojärjestelmän kehittäminen 32 sivua 11.5.2014
Tutkinto	insinööri (AMK)
Koulutusohjelma	sähkötekniikan koulutusohjelma
Suuntautumisvaihtoehto	terveydenhuollontekniikka
Ohjaaja	lehtori Timo Kasurinen

Insinöörityön tarkoituksena oli suunnitella ja toteuttaa kiinteistöhuollon käyttöön soveltuva tietojärjestelmä. Kiinteistöhuollon työt jakautuvat karkeasti kahteen kategoriaan: huoltosopimuksiin sisältyviin töihin ja erikseen laskutettaviin töihin. Insinöörityössä kehitetty järjestelmä keskittyi erikseen laskutettavien töiden tallentamiseen, niihin liittyviin hakuihin sekä raportointiin. Järjestelmään toteutettiin suomenkielinen käyttöliittymä, jotta sen käyttäminen olisi mahdollisimman yksinkertaista. Järjestelmällä pyrittiin vähentämään työmäärää, joka kuluu laskutettavien töiden tietojen tallentamiseen, raportointiin sekä tietojen hakemiseen mahdollisten tiedustelujen yhteydessä.

Järjestelmän pohjaksi valittiin MS SQL Server 2012, joka toimii tietojen tallennuspaikkana sekä haku- ja raportointityökaluna. Suomenkielinen käyttöliittymä toteutettiin C# -ohjelmointikielellä WPF-työpöytäsovelluksena. Järjestelmään kehitettiin lisäksi C# -ohjelmointikielellä toteutettu WCF-palvelu käyttöliittymän ja SQL-palvelimen välille. Palvelun tehtävä oli tietojen ja komentojen välittäminen käyttöliittymän ja SQL-palvelimen välillä. Järjestelmän kehitystyössä käytettiin MS Visual Studio -kehitysympäristöä, jolla voitiin toteuttaa sekä WPF-käyttöliittymä, että WCF-palvelu. Järjestelmän käyttämät tietokannat toteutettiin MS SQL Server Management Studiolla.

Työn lopputuloksena syntyi tietojärjestelmä, jonka suunniteltu käyttöönotto on kesällä 2014. Järjestelmä toteutettiin siten, että sen laajentaminen ja ominaisuuksien lisääminen olisi mahdollisimman yksinkertaista tulevaisuudessa.

Avainsanat	tietojärjestelmä, kiinteistöhuolto

Author	Aleksi Oinas
Title	Information System Development for Property Maintenance
Number of Pages	32 pages
Date	11 May 2014
Degree	Bachelor of Engineering
Degree Programme	Electrical engineering
Specialisation option	Medical Engineering
Instructor	Timo Kasurinen, Senior Lecturer

The purpose of this Bachelor's thesis was to design and implement an information system for use at property maintenance. Tasks in property maintenance can be categorized roughly in two categories: tasks included in maintenance contracts and tasks that are to be invoiced separately. The information system in this project concentrated on storing, searching and reporting functions on tasks that are to be invoiced separately. The user interface of the information system was implemented in Finnish to provide easy operation for users. The purpose of the information system was to reduce the amount of work needed to save task descriptions and generate customer reports. System was also designed and implemented to provide an efficient way to search data from old task descriptions if needed.

The base of the information system was MS SQL Server 2012 Express. The server acted as data storage and also provided search and reporting features for the system. The user interface was programmed in C# programming language as a WPF application. A WCF service was designed and implemented to relay commands and transfer data between the user interface and the SQL server. C# programming language was used also for programming WCF service. The development environment for WPF and WCF was MS Visual Studio. SQL Server Management Studio was used to develop the SQL server databases.

The result of this Bachelor's thesis is an information system that is planned to be deployed in summer of 2014. The system design is such that adding functionalities and properties in the future is made as easy as possible.

Keywords	information system, property maintenance

Sisällys

Lyhenteet

C#	Microsoftin kehittämä ja ylläpitämä ohjelmointikieli.
MVVM	Model-View-Viewmodel. Ohjelmoinnissa käytetty konsepti sovelluksen osioinnista.
SQL	Structured Query Language. Tietokantaohjelmoinnissa käytetty ohjelmointikieli.
SSRS	SQL Server Reporting Services. SQL-palvelimiin saatava raportointilisäosa.
WCF	Windows Communications Foundation. Microsoftin kehittämä ja ylläpitämä kirjasto palvelujen kehittämistä varten.
WPF	Windows Presentation Foundation. Microsoftin kehittämä ja ylläpitämä kirjasto Windows-työpöytäsovellusten graafista toteutusta varten.
XAML	Extensible Application Markup Language. Microsoftin kehittämä, ja ylläpitämä XML-pohjainen kieli. Käyttöliittymän elementtien määrityksiin käytetty kieli.

1 Johdanto

Tässä insinöörityössä käsitellään kiinteistöhuollon käyttöön kehitettyä tietojärjestelmää asiakastietojen, työnkuvausten ja tarvikkeiden sekä asiakkailta laskutettavien töiden tallentamista ja raportointia varten. Tehtyjen töiden riittävän tarkka dokumentointi on olennaista laskutuksen ja töiden jäljitettävyyden kannalta. Kiinteistöhuollossa tehdyt työt jakautuvat karkeasti kahteen kategoriaan: huoltosopimukseen sisältyviin töihin ja erikseen laskutettaviin töihin. Tässä työssä kehitettävää järjestelmää on tarkoitus käyttää erikseen laskutettavien töiden dokumentointiin sekä asiakaskohtaisten kuukausiraporttien tuottamiseen.

Tietojärjestelmään on pystyttävä tallentamaan riittävän kattavat tiedot tehdyistä töistä ja niihin käytetyistä tarvikkeista sekä lisäämään tarpeen mukaan lisätietoja ja liitteitä, esimerkiksi valokuvia tai tekstidokumentteja. Laskutettavien töiden tallentaminen tietojärjestelmään mahdollistaa tehtyjen töiden tarkastelun jälkikäteen, eikä tieto jää pelkästään työntekijän muistin varaan. Järjestelmällä haluttiin vähentää työmäärää, joka kului tehtyjen töiden dokumentointiin ja toisaalta jo aiemmin tehtyjen töiden etsimiseen mahdollisten tiedustelujen tai tarkennuspyyntöjen yhteydessä.

Olennaista järjestelmässä on se, että laskutettavista töistä saadaan tallennettua riittävän kattavat tiedot ja liitettyä niihin tarvittaessa paljonkin lisätietoja. Erityisesti valokuvien ja tekstidokumenttien liittäminen tehtyihin töihin haluttiin sisällyttää järjestelmään. Myös tietojen hakeminen järjestelmästä jälkeenpäin pitää olla mahdollisimman yksinkertaista.

Järjestelmän on oltava mahdollisimman helppokäyttöinen, jotta sitä myös käytettäisiin. Tästä syystä järjestelmään päätettiin toteuttaa Windows-ympäristöstä tuttu ikkunapohjainen suomenkielinen käyttöliittymä mahdollisimman selkein toiminnoin. Käyttöliittymän suunnittelussa ja toteutuksessa pyrittiin huomioimaan loppukäyttäjien tarpeet ja toiveet, sekä pitämään käyttöliittymä mahdollisimman modulaarisena eli erillisistä osista koostuvana. Tällöin käyttöliittymän eri osien muokkaaminen ja mahdollisesti uusien osien lisääminen tulevaisuudessa olisi mahdollisimman yksinkertaista.

2 Tietojärjestelmän kehitysvaiheet

Kehitystyö jaettiin osiin kokonaisuuden hallinnan parantamiseksi. Työ aloitettiin järjestelmän vaatimusmäärittelyn ja rajaamisen tekemisellä. Tässä vaiheessa oli tarkoitus määritellä, mitä ominaisuuksia työssä kehitettävään tietojärjestelmään sisältyisi. Vaatimusmäärittelyn ja rajauksen jälkeen järjestelmän kehitystä jatkettiin suunnitteluvaiheella. Suunnitteluvaiheessa oli tarkoitus suunnitella miten ja millä tekniikalla vaatimusmäärittelyssä ja rajauksessa järjestelmään sisällytetyt ominaisuudet käytännössä toteutettaisiin. Suunnittelussa pyrittiin huomioimaan myös järjestelmän testattavuuteen liittyviä asioita.

Kun suunnittelu oli saatu siihen vaiheeseen, että järjestelmän kokonaisuudesta ja käytettävästä tekniikasta oli riittävän hyvä käsitys, siirryttiin varsinaiseen toteutukseen. Toteutusvaiheessa oli tarkoitus toteuttaa vaatimusmäärittely- ja rajausvaiheessa järjestelmään sisällytetyt ominaisuudet suunnitellulla tavalla. Tietojärjestelmän eri osien testaamista sisällytettiin mahdollisimman paljon toteutusvaiheeseen, jotta mahdolliset ohjelmointivirheet suodattuisivat pois varhaisessa vaiheessa. Testaukseen käytettiin pääasiassa yksikkötestejä ja käyttöliittymän osalta myös käyttäjätestausta. Yksikkötestit olivat tehokas menetelmä ohjelmointivirheiden etsimisessä ja niiden korjaamisessa.

Järjestelmän toteutuksessa hyödynnettiin versionhallintaa. Versionhallinnan avulla pyrittiin takaamaan, että järjestelmä voitiin palauttaa edelliseen versioonsa, mikäli jonkin ominaisuuden lisäämisen tai muokkaamisen yhteydessä ilmenisi suuria ongelmia. Järjestelmässä käytettyyn tietokantaan ei ollut valmista versionhallintatyökalua, eikä sellaisen kehittämistä haluttu sisällyttää tähän työhön, joten tietokannan osalta versionhallinta toteutettiin käyttämällä varmuuskopioita. Tietokantaan tehtävien muutosten, etenkin suurempien muutosten yhteydessä, tietokannasta otettiin täydellinen varmuuskopio ennen muutosten tekemistä. Tällä pystyttiin varmistamaan, että tietokanta saatiin tarvittaessa palautettua muutosta edeltävään tilaansa.

Tietojärjestelmän kehitykseen sisältyivät seuraavat vaiheet:

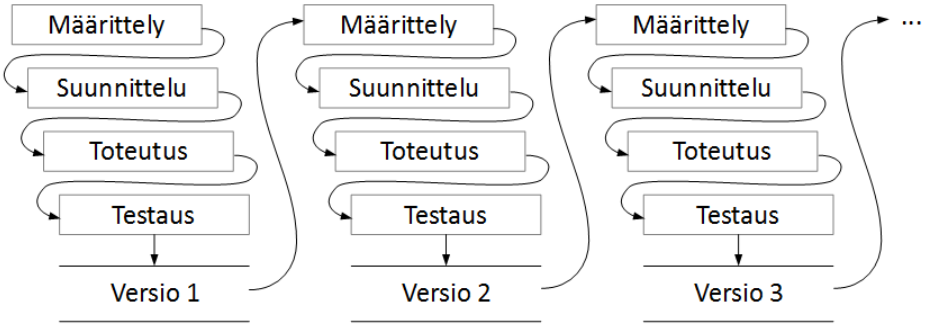
- vaatimusmäärittely ja rajaus
- suunnittelu
- toteutus
- testaus.

Ohjelmistojen kehityksen kuvaamiseen on olemassa useita eri vaihejakomalleja, joilla pyritään kuvaamaan ohjelmiston kehitystyötä tai ohjelmiston koko elinkaarta. Tässä insinöörityössä tehtävän tietojärjestelmän kehitystyö noudatteli ns. Evo-mallia [1, s. 41].

Evo-mallissa kehitysprojekti muodostuu useasta peräkkäisestä kehityssyklistä. Ensimmäisessä syklissä luodaan järjestelmän perusta, jota täydennetään tulevissa sykleissä siten, että jokaisen syklin päätteeksi tietojärjestelmää on laajennettu uudella ominaisuudella, esimerkiksi

> Evo-malli muodostuu sarjasta toistuvia vesiputouksia, joista jokaisen tuloksena on uusilla ominaisuuksilla kasvatettu järjestelmä. [1, s. 41]

Tämä kehitysmalli soveltui hyvin työssä tehtävän tietojärjestelmän kehitysmalliksi, koska tietojärjestelmä oli rakenteeltaan modulaarinen ja sisälsi useita eri ominaisuuksia, joiden tuli toimia yhteisen alustan päällä. Kuvassa 1 esitetään Evo-vaihejakomallin mukainen ohjelmiston kehitystyön eteneminen.



Kuva 1. Evo-vaihejakomallin mukainen ohjelmiston kehitys [1, s. 42]

2.1 Tietojärjestelmän vaatimusmäärittely ja rajaus

Tietojärjestelmän kehittäminen aloitettiin tekemällä järjestelmän vaatimusmäärittely ja rajaus. Tämän vaiheen tarkoituksena oli määritellä, mitä ominaisuuksia tietojärjestelmään sisältyisi ja mitä järjestelmän odotettiin tekevän. Vaiheessa pyrittiin luomaan mahdollisimman kattava yleiskuva järjestelmästä ja sen toiminnoista. Vaatimusmäärittely- ja rajausvaiheessa ei otettu kantaa mahdolliseen toteutustapaan tai toteutuksessa käytettävään tekniikoihin. Vaiheessa keskityttiin kokonaisuuden vaatimusmäärittelyyn ja rajaamiseen, ei niinkään yksityiskohtiin. Yksityiskohtiin ja toteutustapaan perehdyttiin suunnitteluvaiheessa.

Huoltoyhtiön kanssa käytyjen keskustelujen pohjalta tietojärjestelmälle asetettiin seuraavat vaatimukset:

- Järjestelmään piti pystyä tuomaan tiedot aikaisemmin käytössä olleesta MS Access-sovelluksesta.
- Järjestelmän tuli toimia yhdessä tietokoneessa, Windows 7 ja Windows 8 -käyttöjärjestelmillä.
- Järjestelmää tuli voida käyttää suomenkielisellä graafisella käyttöliittymällä.
- Järjestelmään tuli sisältyä tietokanta tietojen tallentamista, hakuja ja raportointia varten.

Järjestelmän yleisten vaatimusten lisäksi graafiselle käyttöliittymälle määriteltiin seuraavat lisävaatimukset. Lisävaatimusten tarkoituksena oli varmistaa, että käyttäjärajapinta täyttäisi mahdollisimman hyvin asiakastarpeet. Käyttöliittymällä piti voida

- lisätä, muokata ja poistaa asiakastietoja
- luoda, muokata ja tulostaa asiakaskohtaisia raportteja
- ylläpitää hinnastoa.

2.2 Tietojärjestelmän toteutuksen suunnittelu

Suunnitteluvaiheessa pyrittiin suunnittelemaan, millä tekniikoilla ja miten vaatimusmäärittely- ja rajausvaiheessa järjestelmään sisällytetyt ominaisuudet tultaisiin toteuttamaan. Suunnittelussa pyrittiin huomioimaan myös järjestelmän testattavuus. Järjestelmän ohjelmointi päätettiin toteuttaa C#-ohjelmointikielellä siten, että järjestelmän eri osien testauksessa voitaisiin hyödyntää mahdollisimman tehokkaasti yksikkötestausta. Suunnitteluvaiheessa ei tehty yksityiskohtaisia suunnitelmia siitä, kuinka esimerkiksi jokin SQL-palvelimen yksittäinen proseduuri tultaisiin toteuttamaan, vaan keskityttiin suunnittelemaan järjestelmän toteutustapa ja eri toiminnot. Tietojärjestelmä haluttiin toteuttaa mahdollisimman modulaariseksi, jotta ominaisuuksien muokkaaminen ja lisääminen tulevaisuudessa olisi mahdollisimman yksinkertaista.

Tietojärjestelmän pääasiallinen tehtävä oli toimia erikseen laskutettavien töiden tallennus-, haku- ja raportointityökaluna. Näin ollen järjestelmään tuli sisältyä useita toisiinsa linkittyviä tietoja, esimerkiksi asiakastiedot ja asiakkaalle tehdyt työt. Tästä syystä tieto-

järjestelmän pohjaksi valittiin SQL-palvelin. Nimestään huolimatta SQL-palvelin ei vaadi erillistä palvelintietokonetta, vaan se voidaan asentaa tietojärjestelmää käyttävään ja ylläpitävään työasemaan.

SQL-palvelimia on saatavilla useita eri vaihtoehtoja. Tietojärjestelmässä käytettävän SQL-palvelimen valintaan vaikuttivat ensisijaisesti hankintahinta, mahdolliset lisenssit sekä yhteensopivuus muun järjestelmän kanssa. Tietojärjestelmää tulisi alkuvaiheessa käyttämään ainoastaan yksi käyttäjä kerrallaan ja SQL-palvelimeen tallennettava tietomäärä kehitettävässä tietojärjestelmässä olisi kohtalaisen pieni. Järjestelmän päätietokannan suurimmat taulukot olisivat täyttönopeudeltaan muutamia tuhansia uusia tietueita per vuosi. Näistä syistä SQL-palvelimen yksittäisen tietokannan maksimikoko tai usean prosessorin tuki eivät muodostuneet rajoittaviksi tekijöiksi palvelinta valittaessa. Koska tietojärjestelmään tuli sisältyä raportointitoiminto asiakaskohtaisten kuukausiraporttien tarkastelua ja tulostusta varten, päädyttiin SQL-palvelimeen, joka sisälsi SSRS-palvelun. SSRS eli SQL Server Reporting Services on MS SQL Server -tuoteperheeseen kuuluva raportointipalvelu, jolla pystytään tuottamaan ja julkaisemaan raportteja suoraan SQL-palvelimella.

Tietojärjestelmän pohjaksi valittiin edellä mainittujen kriteerien perusteella MS SQL Server 2012 Express with Advanced Services. Microsoftin SQL-palvelinten Express-mallit ovat käyttäjälle ilmaisia, mukaan lukien yrityskäyttö. Express-mallien with Advanced Services -versiot sisältävät myös edellä mainitun SSRS-raportointipalvelun. Express-mallit ovat tietokantakooltaan verrattain pieniä, yksittäisen tietokannan maksimi koko on 10 Gt. [2.] Työssä kehitettävän tietojärjestelmän pienestä koosta ja käyttäjämäärästä johtuen Express-malli soveltui hyvin järjestelmän pohjaksi.

Nykyiset SQL-palvelimet pystyvät hallitsemaan helposti työssä kehitettävän tietojärjestelmän tietomäärän, hakutoiminnot sekä raportoinnin. SQL-palvelimen käyttö yksistään esimerkiksi MS SQL Management Studion kautta vaatii käyttäjältä kohtalaisen paljon perehtymistä ja osaamista. Edellä mainittu MS Management Studio onkin pääasiassa tietokannan hallintaan liittyvä työkalu, ei niinkään käyttäjärajapinta.

Jotta SQL-palvelimen tallennus-, haku- ja raportointiominaisuuksia pystyttiin käyttämään halutulla tavalla, järjestelmään kehitettiin suomenkielinen graafinen käyttöliittymä. Käyttöliittymän osalta tärkeimpinä kriteereinä pidettiin suomenkieltä ja helppokäyttöisyyttä. Käyttöliittymän, kuten koko järjestelmän, tuli toimia Windows 7 ja Windows 8 -käyttöjärjestelmillä.

Tietojärjestelmän käyttöliittymä toteutettiin Windows-työasemassa toimivana ns. työpöytäsovelluksena. Toteutustavaksi valittiin C#-ohjelmointikielellä toteutettu WPF-sovellus. WPF eli Windows Presentation Foundation on Microsoftin kehittämä ja ylläpitämä kirjasto, joka muodostaa pohjan Windows-käyttöjärjestelmien graafiselle rajapinnalle [3]. C#-ohjelmointikielessä yhdistyvät MS Visual Basic -ohjelmointikielen helppo syntaksi sekä C++-ohjelmointikielen laskentateho [4, s. 12].

Suomenkielisen graafisen käyttöliittymän tarkoitus oli tarjota käyttäjille selkeä ja yksinkertainen tapa hakea, syöttää ja muokata SQL-palvelimeen tallennettuja tietoja. SQL-palvelimen ja käyttöliittymän väliin kehitettiin erillinen palvelu, joka tekisi käyttöliittymän pyynnöstä hakuja, lisäyksiä ja muutoksia tietokantaan. Palvelu toimisi myös tiedonsiirtoväylänä SQL-palvelimen ja käyttöliittymän välillä tuomalla SQL-palvelimelta tietoja käyttöliittymässä näytettäväksi. Erillisen palvelun käyttäminen käyttöliittymän ja SQL-palvelimen välillä paransi myös tietojärjestelmän laajennettavuutta ja tietoturvaa. Palvelun mahdollisti SQL-palvelimelle tulevien komentojen suodattamisen, jonka ansiosta SQL-palvelimelle ei tullut käyttöliittymältä mitä tahansa komentoja, vaan ainoastaan palvelun kelpuuttamia ja välittämiä komentoja.

Palvelua käyttämällä voitiin estää tehokkaasti SQL-palvelimeen kohdistuvat mahdolliset injektiohyökkäykset. SQL-injektiossa hyökkääjä antaa SQL-palvelimelle komentoja, joita hänen ei pitäisi voida antaa. Palvelua käyttämällä olisi myös yksinkertaista toteuttaa tiedonsiirron suojaaminen. Tietoturvaparannusten lisäksi palvelun käyttäminen tietokannan ja käyttöliittymän välillä helpottaisi mahdollista käyttöliittymän vaihtamista tai uuden käyttöliittymän lisäämistä tulevaisuudessa.

2.3 Tietojärjestelmän osien toteutus

Tietojärjestelmän toteuttaminen aloitettiin tekemällä Evo-kehitysmallin mukaisesti järjestelmän alusta, jolle ruvettiin kokoamaan järjestelmän eri osia. Ohjelmointivaiheessa tietojärjestelmään pyrittiin tekemään Evo-mallia noudattaen aina yksi ominaisuus kerrallaan. Tällöin järjestelmän kokonaisuus pysyi paremmin hallinnassa. Tietojärjestelmän pääasiallisena kehitysympäristönä käytettiin MS Visual Studiota, millä voitiin kehittää sekä käyttöliittymä että palvelu. SQL-palvelimen tietokantojen ja tallennettujen proseduurien kehittämiseen käytettiin SQL Server Management Studiota.

Sekä MS Visual studio että SQL Server Management Studio ovat Microsoftin ohjelmistoja, joten niiden käyttöliittymien ja komentojen kieli on englanti. Tästä syystä myös ohjelmoinnissa päätettiin käyttää englantia. Tällöin esimerkiksi funktiokutsujen tai taulukoiden nimien yhteydessä ei tulisi ongelmia skandinaavisten kirjainten kanssa.

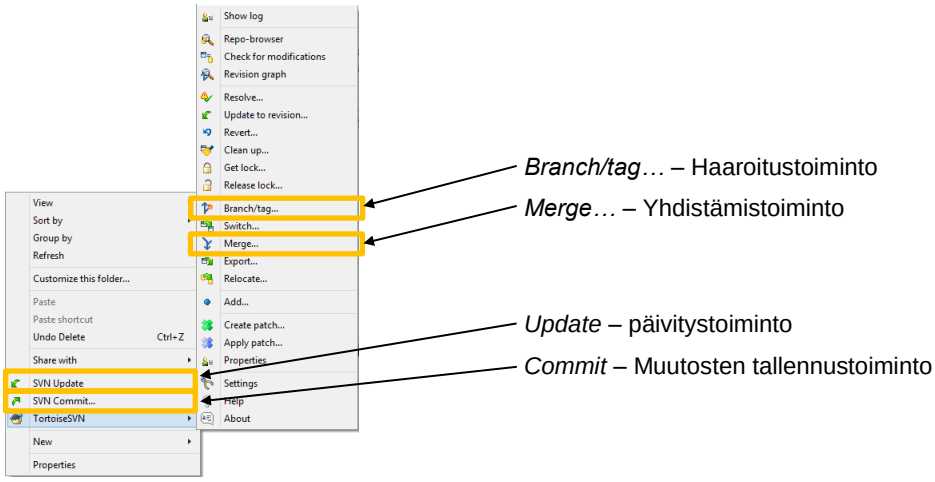
Tietojärjestelmän ohjelmoinnissa hyödynnettiin versionhallintaa, mikä toteutettiin TortoiseSVN-versionhallintatyökalulla. TortoiseSVN on kehitysympäristöstä riippumaton sekä ilmainen käyttää. TortoiseSVN:n käyttäminen ei vaatinut varsinaista versionhallintapalvelinta, vaan koko versionhallinta oli mahdollista toteuttaa käyttämällä pelkästään TortoiseSVN-työkalua. Ilman versionhallintapalvelinta versiomuutoksista ei tullut erillisiä ilmoituksia, vaan käyttäjän tuli huolehtia siitä, että käytössä oli haluttu versio.

Versionhallinnan käyttämisen perustoiminnot ovat *update* ja *commit*. *Update*-toiminnolla tietokoneella oleva ns. paikallinen kopio voidaan päivittää uusimpaan tai johonkin muuhun haluttuun versioon. *Commit*-toiminnolla voidaan puolestaan lähettää paikallisen kopion muuttunut sisältö versionhallinnalle, jolloin se on muiden kehitystyöhön osallistuvien saatavilla. Näiden lisäksi versionhallinnalla voidaan luoda erillisiä kehityshaaroja *branch*-toiminnolla sekä yhdistää luotuja haaroja *merge*-toiminnolla.

Tietojärjestelmän kehityksessä toimintatapa oli se, että aina uuden ominaisuuden kehittämisen aluksi versionhallinnalla luotiin päähaarasta erkaneva kehityshaara. Luodussa kehityshaarassa toteutettiin ominaisuuden kehittäminen ja testaaminen. Kun ominaisuus oli saatu valmiiksi, testattu ja todettu toimivaksi, kehityshaara liitettiin takaisin päähaaraan. Näin versionhallinnan päähaarassa oli koko ajan uusin toimiva versio ja ominaisuuksien lisäykset ja muutokset toteutettiin kehityshaaroissa. Varsinkin uusien ominaisuuksien lisäämisen yhteydessä oli hyödyllistä tehdä ominaisuuksille omat kehityshaarat. Tällöin ominaisuuksien muokkaaminen tai poistaminen jälkeenpäin oli helpommin toteutettavissa. Esimerkiksi jonkin ominaisuuden poistaminen pystyttiin toteuttamaan niin, että järjestelmä palautettiin versionhallinnasta ominaisuuden liittämistä edeltävään versioon. Palauttamisen jälkeen versioon voitiin liittää tarvittaessa muita kehityshaaroja. Näin järjestelmästä saatiin luotua sellainen versio, mistä oli poistettu jokin tietty ominaisuus, mutta joka sisälsi edelleen kaikki muut ominaisuudet. Pienempiä korjauksia ja muutoksia varten ei luotu erillisiä kehityshaaroja vaan korjaukset tehtiin suoraan versionhallinnan päähaarassa.

TortoiseSVN integroitui suoraan Windows-käyttöjärjestelmän hakemistojen kontekstivalikkoon, minkä ansiosta sen käyttäminen oli yksinkertaista. Kuvassa 2 (ks. seur. s.)

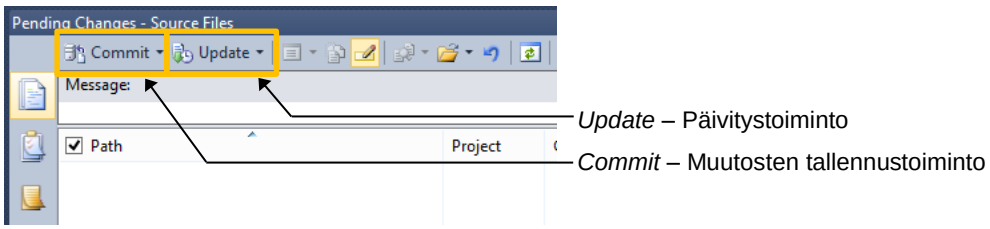
esitetään TortoiseSVN-versionhallintatyökalun perustoimintoja. Toiminnot voitiin suorittaa suoraan halutussa hakemistossa, jolloin *update*- ja *commit*-toiminnot kohdistuivat ainoastaan valittuun hakemistoon ja sen sisältämiin tiedostoihin.



Kuva 2. TortoiseSVN-versionhallinnan perustoiminnot

Versionhallinnan käyttöä helpottamaan ja nopeuttamaan MS Visual Studioon asennettiin AnkhSVN-versionhallintalisäosa. Lisäosan avulla versionhallinnan toiminnot saatiin integroitua MS Visual Studio -kehitysympäristöön. Tällöin ohjelmoinnin aikana ei tarvinnut erikseen käydä tallennushakemistossa valitsemassa *commit*-komentoa muutosten tallentamiseksi, vaan sama pystyttiin tekemään suoraan MS Visual Studiosta. Tästä oli hyötyä etenkin pienempien muutosten ja korjausten yhteydessä, kun muutettu sisältö voitiin tallentaa versionhallintaan suoraan kehitysympäristössä.

Kuvassa 3 (ks. seur. s.) esitetään AnkhSVN-versionhallintalisäosan *commit*- ja *update*-toiminnot. Toiminnot olivat käytettävissä suoraan MS Visual Studion *Pending Changes* -ikkunassa. *Pending Changes* -ikkunassa näkyi koko ajan myös listaus tiedostoista, joiden sisältö oli muuttunut. Listauksen ansiosta varsinkin pienempien muutosten tallentaminen versionhallintaan ei päässyt unohtumaan niin helposti. Lisäosan avulla versionhallintaan tallennettaviin muutoksiin voitiin liittää helposti myös viestejä. Viesteihin pyrittiin kirjoittamaan lyhyt kuvaus siitä, mitä muutoksia versioon oli tehty. Tällä pyrittiin helpottamaan versioiden erottamista toisistaan.



Kuva 3. AnkhSVN-versionhallintalisäosan työkaluvalikko

Versionhallintaa käyttämällä pyrittiin varmistamaan se, että järjestelmä pystyttiin tarvittaessa palauttamaan aikaisempaan versioonsa, mikäli jonkin version yhteydessä ilmenisi suuria ongelmia. Järjestelmään kehitettävälle käyttöliittymälle ja palvelulle luotiin MS Visual Studiossa omat projektinsa, jotka lisättiin versionhallintaan.

2.4 Tietojärjestelmän testaus

Tietojärjestelmän testaukseen käytettiin pääasiassa kehitysympäristönä käytetyn MS Visual Studion yksikkötestejä sekä käyttöliittymän osalta myös käyttäjätestausta. Yksikkötestauksella tarkoitetaan ohjelmakoodin yksittäisen osan tai moduulin testausta. Tästä syystä yksikkötestauksesta käytetään myös nimitystä moduulitestaus. [1, s. 287]. Yksikkötesteillä pyrittiin testaamaan järjestelmän osien toimintaa eri tilanteissa.

Esimerkiksi asiakastietojen hakutoimintoa voitiin testata eri syötteillä ja seurata, miten järjestelmä toimii. Testeillä oli helppo selvittää, toimiiko järjestelmä halutulla tavalla myös niissä tapauksissa, joissa syötteet ovat epäkelpoja. Edellä mainitussa asiakastietoja hakevassa testissä haku voitiin tehdä asiakasnumerolla, jota ei järjestelmässä ole tai siten, että asiakasnumero puuttui kokonaan. Näissä tilanteissa järjestelmän piti toimia oikein ja hallita virhetilanteet niin, että järjestelmän toiminta ei häiriintyisi. Missään tapauksessa järjestelmä ei saisi kaatua.

Yksikkötestejä lisättiin kehityksen aikana ominaisuuksien kehittämisen yhteydessä. Näin testejä kertyi koko kehityksen ajan sitä mukaa, kun järjestelmä kehittyi. Mm. palvelun ja käyttöliittymän ominaisuuksien ja toimintojen testaamisessa voitiin hyödyntää hyvin yksikkötestejä. Käyttöliittymän painonappien ja linkkien toimivuutta voitiin testata yksikkötesteillä ilman käyttäjän osallistumista testaukseen. Näin mahdolliset ohjelmointivirheet ja virhetoiminnot havaittiin ennen varsinaista käyttäjätestausta. Käyttöliittymän osalta käyttäjätestauksen pääpaino olikin toiminnallisuudessa, käytettävyydessä ja mm. ulkoasussa.

Yksikkötesteillä pyrittiin testaamaan mahdollisimman kattavasti käyttöliittymän ja palvelun eri toimintoja. Yksikkötestejä luotaessa pyrittiin ennakoimaan eri tilanteet, joita järjestelmä voisi joutua käsittelemään. Komentojen parametreja saattaisi puuttua, tai niiden arvot olisivat epäkelpoja. Kaikkien mahdollisten variaatioiden testaaminen oli käytännössä mahdotonta. Esimerkiksi asiakastietojen tallennustoiminnon asiakkaan nimi -parametrin kaikkia mahdollisia arvoja oli käytännössä mahdoton sisällyttää testeihin. Tämän tyyppisissä tapauksissa testeissä pyrittiin huomioimaan jokin normaali tapaus ja kaikki ajateltavissa olevat erikoistapaukset. Edellä mainitun asiakastietoja tallentavan toiminnon testauksessa asiakkaan nimi -parametrille annettiin mahdollisimman monia erityyppisiä arvoja, esimerkiksi:

- Arvo, joka täyttää kaikki kelpoisuussäännöt.
- Arvo, joka tulisi hylätä.
- Arvon tilalle annetaan tyhjä merkkijono.
- Arvoksi annetaan jokin muu kuin merkkijono.
- Arvoa ei anneta lainkaan.

Näin toteutettuna testitapaukset kattoivat suuren osan eri tilanteista, joita järjestelmän tulisi pystyä käsittelemään.

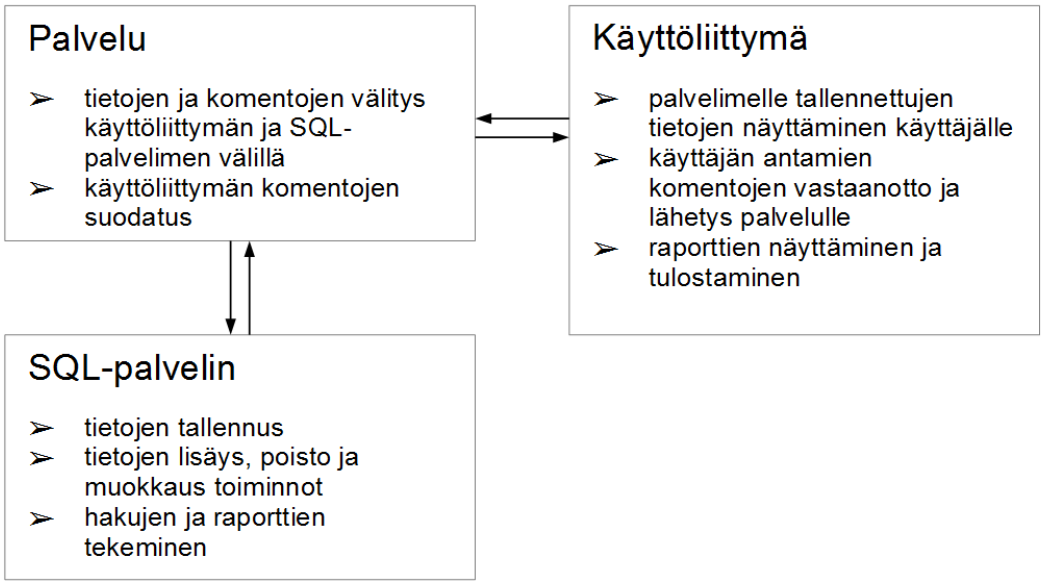
Hyödyntämällä yksikkötestausta järjestelmän eri toimintojen testaaminen oli huomattavasti nopeampaa, kuin komentojen testaaminen käsityönä yksi komento kerrallaan. Käsityönä tehtävässä testauksessa testien välillä voisi olla suuriakin eroja ja testitapauksia voisi unohtua. Käyttämällä yksikkötestejä, testit suoritettiin aina samalla tavalla ja testeihin sisältyi jokaisella testikerralla samat testitapaukset.

3 Tietojärjestelmän toteutus

Tietojärjestelmän toteutuksessa pyrittiin kehittämään aina yksi ominaisuus kokonaisuudessaan valmiiksi ennen seuraavaan ominaisuuteen siirtymistä. Tällä tavalla kokonaisuus pysyi paremmin hallinnassa ja kaikki ominaisuuteen liittyvät toiminnot valmistuivat suunnilleen samaan aikaan. Aina, kun jokin ominaisuus kehitettiin esimerkiksi käyttöliittymään, kehitettiin samalla myös siihen liittyvät toiminnot palveluun ja SQL-palvelimelle. Tästä oli hyötyä etenkin ominaisuuksien testaamisessa, koska ominaisuuksien valmistuttua kaikki niiden tarvitsemat toiminnot olivat käytettävissä.

Tämän ansiosta yhdellä testillä voitiin testata sekä yksittäisen ominaisuuden toimintaa että tietojen ja komentojen siirtymistä järjestelmän eri osien välillä.

Kuvassa 4 esitetään tietojärjestelmän eri osia sekä niiden pääasialliset tehtävät. Kuvan nuolet esittävät tietojen ja komentojen liikkumista järjestelmän eri osien välillä.



Kuva 4. Tietojärjestelmän osat ja niiden pääasialliset tehtävät

3.1 SQL-palvelimen tietokantojen toteutus

Tietojärjestelmän pohjana ja tietojen tallennuspaikkana toimi SQL-palvelin. Työssä kehitetyn tietojärjestelmän käyttämä SQL-palvelin oli Microsoftin MS SQL Server 2012 Express with Advanced Services. Advanced Services -versio valittiin sen takia, että tietojärjestelmään liittyvät raportointiominaisuudet haluttiin toteuttaa SQL-palvelimella julkaistavina SSRS-raportteina. Raporttien toteuttaminen SSRS-raportteina mahdollisti sen, että raportteja pystyttiin tarvittaessa tarkastelemaan ja tulostamaan verkkoselaimella suoraan SQL-palvelimelta. Tämä lisäsi järjestelmän käyttövarmuutta mikäli palvelun tai käyttöliittymän toiminnassa ilmenisi suuria ongelmia.

Tietojärjestelmän SQL-palvelimeen päätettiin tehdä kaksi tietokantaa: päätietokanta ja lokitietokanta. Päätietokanta toimi tietojen varsinaisena tallennuspaikkana ja sisälsi kaikki tarvittavat taulukot ja proseduurit järjestelmän normaalia käyttöä varten. Lokitietokanta toimi loki- ja virhetietojen tallennuspaikkana. Lokitietokanta sisälsi tarvittavat

taulukot ja proseduurit virheilmoitusten ja poikkeusten tallentamista ja hakemista varten. Varsinkin järjestelmän kehitysvaiheessa virheiden ja poikkeusten tallentaminen helpotti huomattavasti vikojen paikallistamista ja niiden korjaamista. Ilman lokitietoja virhetilanteet olisi pitänyt yrittää toistaa ja tutkia samalla, mikä virheen aiheutti.

Lokitietojen tallentamiseen tarkoitettu tietokanta oli rakenteeltaan hyvin yksinkertainen. Tietokanta sisälsi ainoastaan taulukot yleisiä lokitietoja, C#-osan virheitä sekä SQL -palvelimen virheitä varten. Taulukoissa ei käytetty viiteavaimia eli taulukoiden väleille ei muodostettu minkäänlaisia relaatioita. Rakenne pidettiin sellaisena, että taulukot olivat ainoastaan tietojen tallennuspaikkoja eikä tietokannalla ollut tarkoituskaan tehdä mitään monimutkaisia hakuja taulukoiden välillä. Taulukoille määritettiin ainoastaan halutut sarakkeet, sarakkeiden tietotyypit sekä laskurityyppiset pääavainsarakkeet. Laskurisarakkeilla varmistettiin se, että taulukoiden rivit eli tietueet pysyivät yksilöllisinä.

Yleisten lokitietojen tarkoituksena oli se, että järjestelmän tekemiä toimintoja voitiin tallentaa ja tarkastella esimerkiksi komentojen suoritusjärjestystä. Yleiset lokitiedot helpottivat huomattavasti virheiden paikantamista ja niiden korjaamista. Lokitietojen avulla voitiin nopeasti selvittää, mitä järjestelmä oli tehnyt ennen virhetilannetta ja mitä sen jälkeen.

Virhetietoja sisältävien taulukoiden oli tarkoitus toimia virheiden ja poikkeusten tallennuspaikkana. Virheiden tallentaminen oli jaettu kahteen taulukkoon. Virhetietojen tallennustaulukko määräytyi sen perusteella, tapahtuiko virhe SQL-palvelimella vai C# -osassa järjestelmää. Jakamalla tiedot kahteen taulukkoon, pystyttiin nopeasti rajaamaan virheiden tapahtumapaikat.

SQL-palvelimen päätietokanta pyrittiin myös pitämään mahdollisimman yksinkertaisena. Tietokantaan luotiin tarvittavat taulukot sekä taulukoiden viiteavaimet. Taulukoihin luotiin myös laskurityyppiset perusavainsarakkeet, jotka toimivat tietueiden yksilöllisinä tunnisteina. Päätietokantaan luotiin kaikki tietojärjestelmän normaalissa käytössä tarvittavat proseduurit. Proseduureja käytettiin tietojen hakemiseen, lisäykseen, muokkaamiseen ja poistamiseen. Päätietokantaan luotiin myös kaikki raportointiin liittyvät proseduurit.

Viiteavaimia käyttämällä taulukoiden välille muodostettiin relaatioita. Relaatio voitiin toteuttaa esimerkiksi siten, että viiteavaimena toimivan sarakkeen arvoille piti löytyä vastineet viitatun taulukon perusavainsarakkeesta. Viiteavainten luomisen yhteydessä

määriteltiin myös toiminta niissä tapauksissa, kun taulukon tietueita muokataan tai poistetaan. Työssä kehitetyssä tietojärjestelmässä yleisin toiminta oli ns. *kaskadi*, jossa tietueiden muutokset ja poistot pääavaimen sisältävässä taulukossa päivittyivät automaattisesti viiteavaimen sisältävään taulukkoon. *Kaskadia* käyttämällä taulukoihin ei jäänyt ns. orpoja tietueita joiden viiteavain ei viitannut mihinkään, eikä tietueita joiden viiteavain olisi osoittanut väärään pääavaimeen.

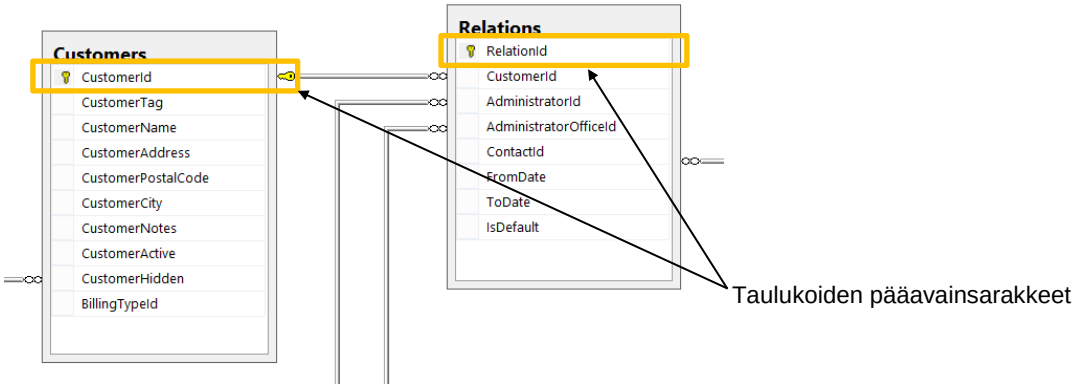
Kaskadia käytettäessä SQL-palvelin huolehti automaattisesti siitä, että viittaukset pysyivät eheinä. *Kaskadin* käyttäminen ei ole mahdollista, jos taulukoiden välillä on jo ennestään *kaskadi*-tyyppinen relaatio. Tällöin *kaskadin* käyttäminen voisi aiheuttaa ns. silmukkaviittauksen, jossa tietueiden poistot tai päivitykset aiheuttavat silmukan, joka jää päivittämään tai poistamaan tietueita loputtomasti. Tietojärjestelmän taulukoiden välisiä relaatioita luotaessa kiinnitettiin erityistä huomiota mahdollisiin silmukkaviittauksiin. Tästä syystä taulukoiden väleille pyrittiin muodostamaan ainoastaan yksittäisiä relaatioita, jolloin silmukkaviittaukset eivät olisi mahdollisia.

Riippuen taulukon sisältämistä tiedosta, taulukoihin määriteltiin indeksejä hakujen tehostamiseksi. Taulukoihin pyrittiin lisäämään vain tarvittavat indeksit. Ne lisättiin sellaisiin sarakkeisiin, joita käytettiin yleisimmin hakukriteereinä. Tällaisia sarakkeita olivat esimerkiksi tehtyjen töiden päivämääräsarakkeet sekä asiakasnumerot. Sopivien indeksien lisääminen taulukoihin nopeutti niihin tehtäviä hakuja.

Indeksoimattomaan taulukkoon tehtävässä haussa SQL-palvelin joutuu käymään taulukon rivi riviltä läpi etsiessään hakukriteerit täyttäviä tietueita. Vastaavasti indeksoituun taulukkoon tehtävässä haussa SQL-palvelin pystyy rajaamaan haun tiettyihin tietueisiin, eikä sen näin ollen tarvitse käydä läpi kaikkia taulukon tietueita. Indeksien avulla hakuajat lyhenivät ja samalla koko tietojärjestelmän suorituskyky parani. SQL-palvelimen taulukoita ja indeksejä suunniteltaessa pyrittiin huomioimaan taulukoihin tehtävät tyypillisimmät haut ja niiden käyttämät hakukriteerit. Indeksejä luotaessa oli myös muistettava, että ylimääräiset indeksit hidastaisivat taulukoihin tehtäviä lisäyksiä ja kasvattaisivat turhaan tietokannan kokoa.

Kuvassa 5 (ks. seur. s.) esitetään osa tietojärjestelmän asiakastietojen tallennukseen liittyvästä taulukkorakenteesta ja taulukoiden välisiä relaatioita. Taulukoiden väliset relaatiot ovat tyypiltään yksi-moneen -relaatioita. Tällä tarkoitetaan sitä, että yksittäisellä pääavaimen arvolla voi olla useita vastineita viiteavainsarakkeessa. Kuvan taulukko-

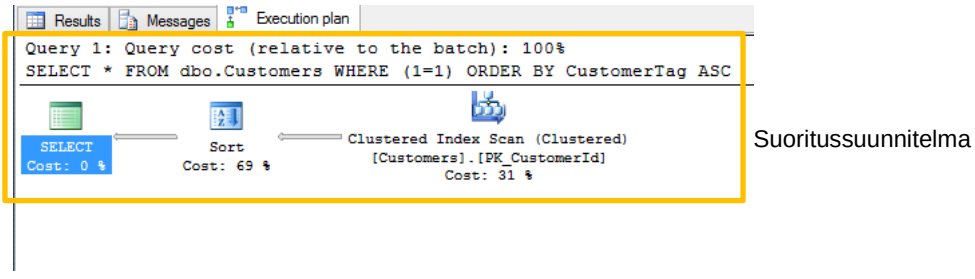
rakenteessa yhdellä asiakasnumerolla (CustomerId), voi olla useita vastineita Relations-taulukon viiteavainsarakkeessa (CustomerId).



Kuva 5. SQL-palvelimen taulukoiden välisiä yhteyksiä

Tietojärjestelmän käyttämät haku-, lisäys-, muokkaus- ja poistotoiminnot toteutettiin SQL-palvelimeen tallennetuilla proseduureilla. Proseduureja kehitettäessä pyrittiin huomioimaan mahdolliset viite-eheysongelmat ja ratkaisemaan ne jo kehitysvaiheessa. Proseduurien luomiseen käytettiin MS SQL Server Management Studiota, jonka avulla proseduureja oli yksinkertaista luoda, testata ja tarkastella niiden suoritusstatistiikka.

MS SQL Management Studioon sisältyy työkalu, jolla voidaan tarkastella proseduurin suorituksen statistiikkaa ja suoritussuunnitelmaa eli *actual execution plania*. Suoritussuunnitelmasta nähdään mm. proseduurin suoritukseen kulunut aika sekä tehtyjen luku- ja kirjoitusoperaatioiden lukumäärät. Suoritussuunnitelma antaa lisäksi suosituksia indeksien muutoksista, mikäli näillä voitaisiin parantaa proseduurin suorituskykyä [5]. Kuvassa 6 esitetään SQL Management Studion suoritussuunnitelma asiakastietoja hakevalle proseduurille.



Kuva 6. SQL-palvelimen suoritussuunnitelma

Kuvasta 6 (ks. ed. s.) nähtiin, että esimerkkiproseduuriin kokonaissuoritustehosta 69 % kului *Sort*-operaatioon ja jäljelle jäänyt 31 % *Index Scan* -operaatioon. *Sort*-operaatiossa Customers-taulukon CustomerTag-sarakkeen arvoja järjestetään aakkosjärjestykseen. *Index Scan* -operaatiossa asiakasnumerosarakkeesta haetaan hakukriteerin täyttäviä arvoja.

Suoritussuunnitelmaa tarkastelemalla nähtiin, jos jokin proseduuri tai sen osa vaati poikkeuksellisen paljon suoritustehoa. Suoritussuunnitelmasta saatavien tietojen avulla proseduureja voitiin optimoida. Proseduurien suoritusajat pyrittiin pitämään mahdollisimman lyhyinä, koska niiden suoritusnopeus vaikutti suoraan tietojärjestelmän kokonaisnopeuteen. Tietojärjestelmän käyttämä palvelu oli toteutettu siten, että se välitti SQL-palvelimelle käyttöliittymältä vastaanottamansa komennon ja jäi odottamaan vastausta. Palvelun piti siis odottaa proseduurin valmistumista ennen seuraavan toiminnon aloittamista. Tästä syystä SQL-palvelimeen tallennettujen proseduurien sekä taulukoiden indeksien suunnitteluun, optimointiin ja testaukseen kiinnitettiin paljon huomiota.

Käyttämällä SQL-palvelimeen tallennettuja proseduureja, toimintoja pystyttiin korjaamaan ja muokkaamaan ilman, että käyttöliittymää tai palvelua tarvitsi päivittää. Jos toiminnot olisi ohjelmoitu suoraan palvelun tai käyttöliittymän C#-osaan, muutosten tekeminen olisi vaatinut koko järjestelmän päivittämisen. Tämä oli hyödyllinen ominaisuus etenkin pienempien korjausten ja muutosten yhteydessä. Tarvittavat muutokset voitiin toteuttaa tietojärjestelmän ollessa käynnissä ja muutokset tulivat voimaan välittömästi. Jos tallennettujen proseduurien parametreja tai niiden palauttamien taulukoiden rakennetta jouduttiin muuttamaan, piti muutoksen yhteydessä päivittää myös järjestelmän muut osat.

3.2 Palvelun toteutus

Palvelun tehtävänä oli toimia SQL-palvelimen ja käyttöliittymän välisenä rajapintana. Palvelulla toteutettiin yhteys SQL-palvelimen ja käyttöliittymän välille siten, että SQL-palvelin ei ollut suoraan yhteydessä ulkomaailmaan. Palvelu toimi tietojen ja komentojen välittäjänä. Yksinkertaistettuna palvelun tehtävä oli kuunnella käyttöliittymältä komentoja, välittää ne SQL-palvelimelle ja lähettää SQL-palvelimelta tullut vastaus käyttöliittymälle. Palvelun avulla oli myös yksinkertaista toteuttaa tietojärjestelmään kirjautumistoiminto. Kirjautumisessa tietojärjestelmä tunnistaa käyttäjän ja joko sallii käytön tai estää pääsyn tietojärjestelmään.

Kaikki kommunikaatio ja tiedonsiirto käyttöliittymän ja palvelun välillä päätettiin toteuttaa käyttämällä suojattua yhteyttä. Kommunikaation suojaamisella ja tietojärjestelmän kirjautumisominaisuudella haluttiin mahdollistaa yksinkertaisempi siirtyminen hajautetumpaan järjestelmään. Hajautetussa järjestelmässä SQL-palvelin ja palvelu sijaitsisivat erillisellä palvelimella ja käyttöliittymä toisella tietokoneella, mahdollisesti lähiverkon ulkopuolelta. Järjestelmää käytettäessä lähiverkon ulkopuolella esimerkiksi salasanojen tai asiakastietojen lähettäminen suojaamattomalla yhteydellä ei tulisi kysymykseen.

Tietojärjestelmän palvelu toteutettiin WCF-palveluna, jonka ohjelmointiin käytettiin C#-ohjelmointikieltä. WCF eli *Windows Communications Foundation* on Microsoftin kehittämä ja ylläpitämä kirjasto, joka sisältää kaikki tarvittavat ominaisuudet tietojärjestelmässä käytetyn palvelun toteuttamiseen. WCF tarjosi myös valmiudet käyttää palvelua tulevaisuudessa muiden käyttöliittymien kanssa. [6, s. 3].

Tietojärjestelmään kehitetty palvelu toimi siten, että käyttöliittymän lähettämät komennot tarkistettiin ja virhetilanteissa käyttöliittymälle lähetettiin virheilmoitus. Tarkistuksen jälkeen palvelu lähetti komennot edelleen SQL-palvelimelle ja jäi odottamaan vastausta. Kun palvelu sai vastauksen SQL-palvelimelta, vastaus välitettiin takaisin käyttöliittymälle. Tämäntyyppisellä komentojen tarkistuksella ja suodatuksella parannettiin tietojärjestelmän luotettavuutta ja tietoturvaa. Virheelliset komennot eivät pysäyttäneet palvelua, eivätkä ne päätyneet SQL-palvelimelle asti. Virheellisiin komentoihin vastattiin virheilmoituksilla ja virheiden tiedot tallennettiin lokitietokantaan.

Tietojärjestelmän erilaiset hakutoiminnot olivat tyypillisesti sellaisia, että palvelu välitti komennon SQL-palvelimelle ja jäi odottamaan haun valmistumista. Kun SQL-palvelin oli saanut haun tehtyä, palvelu lähetti käyttöliittymälle haun tuloksen. Ohjelmoinnin selkeyttämiseksi palvelun komennot ja komentojen parametrit nimettiin samoin kuin niitä vastaavat SQL-palvelimelle tallennetut proseduurit ja niiden parametrit. Tällä nimeämiskäytännöllä myös virheiden etsiminen ja niiden korjaaminen helpottui, koska palvelun komentoa vastaava SQL-proseduuri voitiin tunnistaa nopeasti nimen perusteella.

Palveluun luotiin joukko luokkia, joita käytettiin tiedonsiirtoon ja viestien lähettämiseen. Työssä kehitetyn palvelun lähettämä perusviesti ilman varsinaista tietojen siirtoa oli tyypiltään luokka, joka sisälsi yhden kokonaisluvun ja yhden merkkijonon. Vastaus saattoi olla esimerkiksi 0 OK tai -1 Virhe. Viestien numerointikäytäntönä pidettiin sitä,

että perusviestit sisälsivät aina positiivisen kokonaisluvun tai nollan. Virheet vastaavasti sisälsivät aina negatiivisen kokonaisluvun.

Koska tietojärjestelmässä saattoi esiintyä kahden tyyppisiä virheitä, päätettiin myös virheilmoitukset numeroida yhtenäisellä tavalla. Erityyppisten virheiden tunnistamiseen käytettiin aina seuraavia kokonaislukuja:

- -1, virhe
- -2, poikkeus.

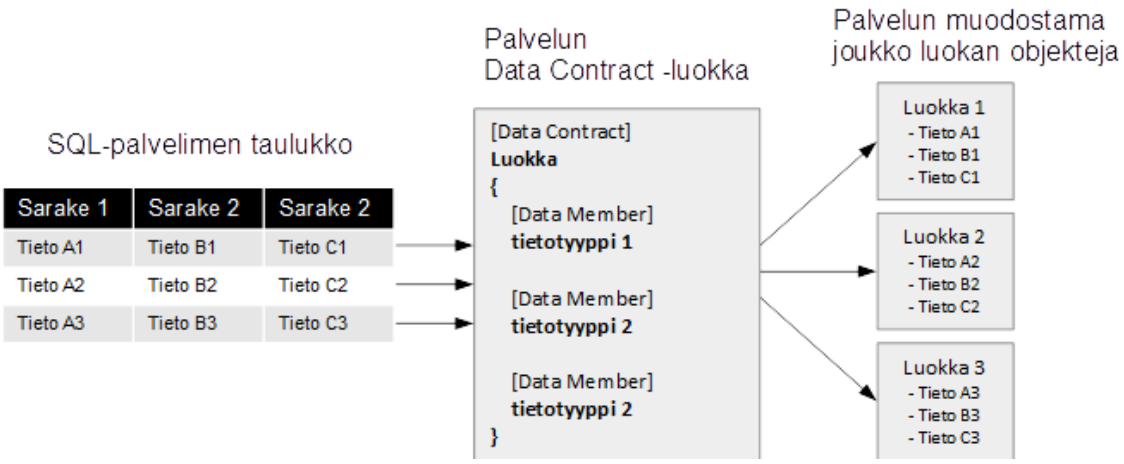
Yhtenäisellä numeroimiskäytännöllä viestien tulkitseminen oli yksinkertaista. Numeroimiskäytäntö mahdollisti myös sen, että virheiden kirjoittaminen tietojärjestelmän lokitietokantaan oli yksinkertaista toteuttaa. Numeroimiskäytännön avulla viestien laatu voitiin tulkita jo pelkän viestin sisältämän kokonaisluvun perusteella.

Virheiden kirjaamisen lisäksi yhtenäisen numeroinnin käyttäminen helpotti huomattavasti virheilmoitusten näyttämistä käyttöliittymässä. Viestin sisältämän kokonaisluvun perusteella käyttöliittymä tunnisti oliko kyseessä virhe vai jokin muu viesti. Virheilmoitusten tapauksissa viestin merkkijono sisälsi tyypillisesti virheilmoituksessa näytettävän tekstin.

Varsinaisten tietojen siirtämiseen käyttöliittymän ja palvelun välillä luotiin luokkia, jotka sisälsivät ominaisuudet siirrettäville tiedoille. Luokkien ominaisuuksien tietotyypit, esimerkiksi kokonaisluku tai merkkijono, määräytyivät sen perusteella, mitä SQL-palvelimen tietoja luokalla oli tarkoitus siirtää. Luokat pyrittiin nimeämään samoin kuin SQL-palvelimen taulukot joiden tietoja luokalla oli tarkoitus siirtää. Esimerkiksi SQL-palvelimen Customers-nimisen taulukon riviä vastaava tiedonsiirtoon käytetty luokka oli nimeltään CustomerDC. Pääte DC lisättiin luokan nimeen selkeyttämään, että kyseessä on tiedon siirtoon tarkoitettu luokka eli ns. *data contract*.

WFC-palvelu käyttää tiedon siirtoon *data contracteja*, jotka sisältävät luokasta riippuen yhden tai useamman *data member* -ominaisuuden [6, s. 103]. Tietojärjestelmän data *contract* -luokat kuvasivat lähetettävän taulukon rivejä ja vastaavasti *data member* -ominaisuudet kyseisen rivin yksittäisiä soluja.

Kuvassa 7 (ks. seur. s.) esitetään SQL-palvelimen taulukon ja tiedon siirtämiseen käytettävän *data contract* -luokan välistä analogiaa.



Kuva 7. SQL-palvelimen taulukon ja palvelun *data contract* -luokan välinen analogia

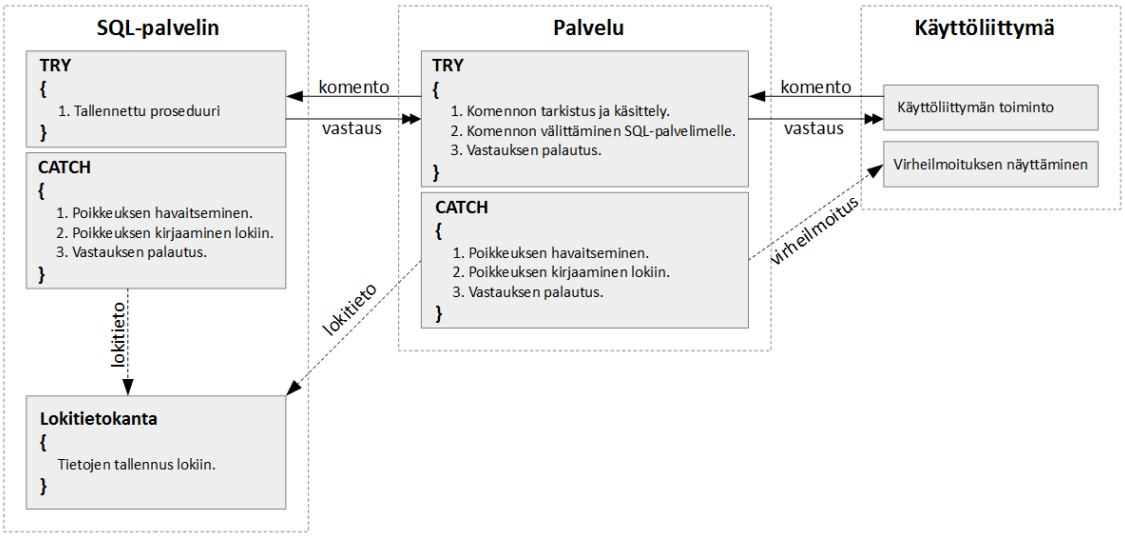
Koska SQL-palvelin palautti useissa tapauksissa tietoja taulukkomuodossa, palvelun piti muuntaa taulukon rivit tiedonsiirtoon käytettäviksi *data contract* -luokiksi. Etenkin erityyppiset haut olivat tyypillisesti sellaisia, että SQL-palvelin palautti hakutuloksen taulukkomuodossa. Jos SQL-palvelimen palauttama taulukko sisälsi useamman kuin yhden rivin, piti palvelun muuntaa erikseen jokainen rivi. Palvelun piti siis tehdä siirrettäville tiedoille useita muunnoksia SQL-palvelimen ja käyttöliittymän välisessä tiedonsiirrossa.

Palvelun tekemien muunnosten suuren lukumäärän ja niissä mahdollisesti tapahtuvien virheiden takia, tietojen muuntaminen taulukkomuodosta *data* contract -luokiksi toteutettiin *Try-Catch* -rakenteen sisällä. *Try-Catch* -rakenne on poikkeuksien havaitsemiseen ja käsittelemiseen käytettävä rakenne. *Try-Catch* -rakenne sisältyy useisiin eri ohjelmointikieliin mm. tietojärjestelmän ohjelmointiin käytettyyn C#-ohjelmointikieleen sekä SQL-palvelimen tallennettujen proseduurien ohjelmointiin käytettyyn SQL-kyselykieleen.

Rakennetta käytettäessä komento, joka voisi aiheuttaa poikkeuksen, suoritettiin *Try*-osan sisällä. Mikäli komento aiheutti poikkeuksen, suoritus siirtyi *Catch*-osaan, missä poikkeus voitiin käsitellä hallitusti. Rakennetta käyttämällä suorituksen aikaiset poikkeukset voitiin havaita ja niihin pystyttiin reagoimaan. Poikkeukset voitiin esimerkiksi kirjata lokitietokantaan tai näyttää virheilmoitus. Useissa tapauksissa järjestelmä suoritti molemmat toiminnot: näytti virheilmoituksen ja kirjasi virheen lokitietokantaan.

Varsinkin palvelun toiminnassa oli tärkeää, että poikkeukset käsiteltiin hallitusti, koska hallitsemattomina ne voisivat pahimmillaan pysäyttää koko palvelun. Poikkeusten

havaitseminen ja niiden käsittely paransivat huomattavasti palvelun toimintavarmuutta. Kuva 8 esittää tietojen ja komentojen liikkumista tietojärjestelmän eri osien välillä sekä tietojärjestelmän eri osien *Try-Catch* -rakenteita.



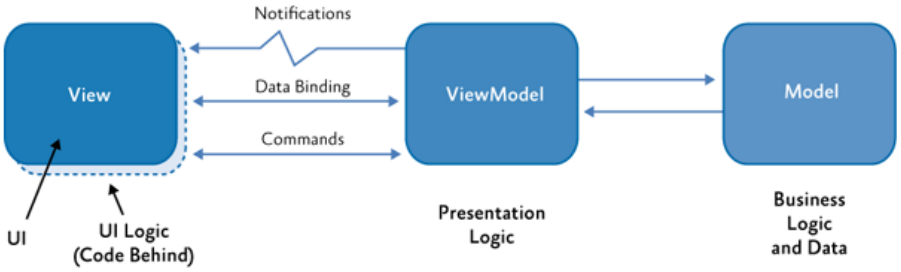
Kuva 8. Tietojen ja komentojen liikkuminen sekä *Try-Catch* -rakenteet järjestelmän eri osissa

3.3 Käyttöliittymän toteutus

Tietojärjestelmän käyttöliittymäksi kehitettiin C#-ohjelmointikielellä toteutettu WPF-työpöytäsovellus. Ohjelmointitapa noudatteli ns. MVVM-konseptia. Siinä käyttöliittymä on jaettu kolmeen osaan:

- malli (*Model)*
- näkymä *(View)*
- näkymämalli (*ViewModel).* [8].

Konseptia hyödyntämällä käyttöliittymän lähdekoodi pysyi selkeästi ryhmiteltynä, minkä ansiosta toimintojen ja ominaisuuksien löytäminen oli yksinkertaista. Lähdekoodi pysyi myös helpommin luettavana. Kuvassa 9 (ks. seur. s.) esitetään MVVM-konseptin mukaisen käyttöliittymän eri osat sekä tietojen, tapahtumien ja komentojen liikkumista eri osien välillä.



Kuva 9. MVVM-konseptin mukaisen käyttöliittymän eri osat [9]

Yhteydet käyttöliittymän eri osien välillä oli toteutettu hyödyntämällä WPF-kirjaston datasidontaa *(Data Binding)*. Datasidonnalla tarkoitetaan sitä, että näkymässä olevat elementit, kuten tekstikentät, on sidottu suoraan näkymämallin ominaisuuksiin. Datasidonta piti huolen siitä, että ominaisuuksien muutokset päivittyivät näkymän ja näkymämallin välillä. Tällöin esimerkiksi näkymässä olevan tekstikentän arvon muuttuessa myös siihen sidottu näkymämallin ominaisuuden arvo muuttui. Datasidontaa hyödyntämällä käyttöliittymän kaikki toiminnot voitiin toteuttaa näkymämalleissa ja malleissa. Näin toteutettuna näkymän ainoaksi tehtäväksi jäi käyttöliittymän elementtien näyttäminen ja näkymän päivittäminen.

Mallien (*Model*) tarkoitus oli toimia tietojärjestelmässä käsiteltävien tietojen tallennuspaikkana sekä kelpoisuussääntöjen tarkistuksiin liittyvien toimintojen tallennuspaikkana. Mallit olivat yksinkertaisia luokkia jotka sisälsivät ominaisuudet tallennettaville tiedoille sekä *IDataErrorInfo*-liittymän ominaisuudet.

IDataErrorInfo on Microsoftin .NET-kirjastoon sisältyvä liittymä, joka sisältää toiminnot virheinformaation näyttämiseen käyttöliittymässä [10]. *IDataErrorInfo*-liittymää hyödyntämällä mallien kelpoisuussääntöjen tuottamat virheilmoitukset voitiin sitoa datasidonnan avulla suoraan näkymän elementteihin.

Sisällyttämällä kelpoisuussäännöt malleihin käyttöliittymällä voitiin toteuttaa helposti palvelulle lähetettävien tietojen tarkistuksia ennen varsinaista tietojen siirtoa. Esimerkiksi tietojärjestelmän käyttämä asiakkaan malli sisälsi ominaisuudet mm. asiakkaan nimelle, katuosoitteelle, postinumerolle ja postitoimipaikalle. Näiden kelpoisuussääntöinä olivat mm. seuraavat:

- Postinumeron piti koostua viidestä numerosta.
- Asiakkaan nimi ei saanut puuttua.

Tarkistusten avulla voitiin estää virheellisten tai puutteellisten tietojen lähettäminen. Näin käyttöliittymä ei kuormittanut palvelua ja SQL-palvelinta lähetyksillä, joiden tallentaminen ei tulisi onnistumaan. Puutteelliset tai virheelliset tiedot hylättäisiin joko palvelussa tai viimeistään SQL-palvelimella.

Kelpoisuussääntötarkistusten tekeminen palvelussa tai SQL-palvelimella olisi vaatinut tarkistettavien tietojen ja tarkistusten tulosten lähettämisen järjestelmän eri osien välillä. Tietojen lähettäminen järjestelmän eri osien välillä olisi hidastanut käyttöliittymän toimintaa ja siten heikentänyt sen käytettävyyttä. Toteuttamalla kelpoisuussääntötarkistukset käyttöliittymässä tarkistuksiin kuluva aika pysyi mahdollisimman lyhyenä. Kelpoisuussääntötarkistusten tekeminen käyttöliittymässä vähensi myös huomattavasti tietojen siirtoa tietojärjestelmän eri osien välillä.

Kelpoisuussääntöjä ja datasidontaa hyödyntämällä voitiin sitoa esimerkiksi asiakastietonäkymän tallennuspainikkeen aktiivisuus eli *IsEnabled*-ominaisuus mallin kelpoisuutta kuvaavaan ominaisuuteen *IsValid*. Tällöin tallennuspainike oli aktiivinen ainoastaan silloin, kun mallin ominaisuudet täyttivät kelpoisuussäännöt. Kuvassa 10 esitetään käyttöliittymän asiakasmallin rakennetta.

```
using System;
using System.ComponentModel;
using WpfClient.Communication.DataService;
using WpfClient.Core;

namespace WpfClient.Models
{
    public class CustomerModel : IDataErrorInfo
    {

        // Creation
        Creation


        // Properties
        Properties


        // IDataErrorInfo Members
        IDataErrorInfo Members


        // Validation
        Validation
    }

}
```

Kuva 10. Asiakasmallin rakenne

Tietojärjestelmän kaikki mallit noudattelivat kuvassa 10 esitettyä rakennetta. Jokaisessa mallissa oli sille tyypillisten ominaisuuksien lisäksi *IDataErrorInfo*-liittymän ominaisuudet sekä tietojen kelpoisuussääntöjen tarkistuksiin liittyvät toiminnot.

C#-ohjelmakoodin luettavuuden ja selkeyden takia kaikki mallit luotiin samalla tyylillä ja mallien eri osat ryhmiteltiin niiden käyttötarkoituksen perusteella. Ryhmät nimettiin kaikissa malleissa samalla periaatteella. Kuvassa 10 (ks. ed. s.) esitetyn asiakasmallin ominaisuudet on ryhmitelty neljään ryhmään:

- *Creation*, luokan luomiseen ja poistamiseen liittyvät toiminnot
- *Properties*, luokan ominaisuudet
- *IDataErrorInfo Members*, kelpoisuussääntöihin liittyvät ominaisuudet
- *Validation*, kelpoisuussääntöjen tarkistuksiin liittyvät toiminnot.

Kaikki käyttöliittymässä näytettävä sisältö sekä komentojen toiminnallisuus oli toteutettu näkymämalleissa *(ViewModel)*. Näkymämallien pohjana toimi pääasiassa malli, jonka ominaisuuksia oli tarkoitus näyttää, lisätä, muokata tai poistaa käyttöliittymällä.

Yhteys näkymämallin ja näkymän välillä oli toteutettu MVVM-konseptin mukaisesti hyödyntämällä WPF-kirjaston datasidontaa (*Data Binding*) sekä ilmoituksia (*Notifications*). Ilmoitusten tehtävänä oli välittää näkymälle tieto näkymämallin muutoksista.

Tietojärjestelmän näkymämallit kehitettiin siten, että kaikki näkymämallit periytyivät abstraktista perusmallista ViewModelBase. Perusmalli sisälsi seuraavat liittymät:

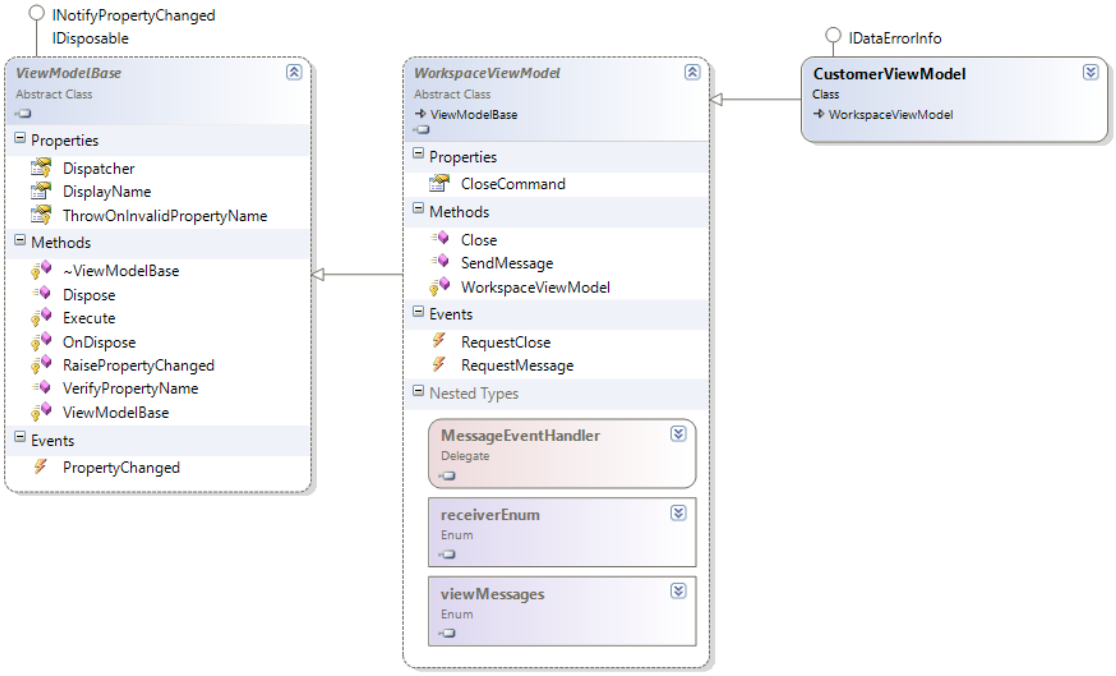
- *INotifyPropertyChanged*
- *IDisposable.*

INotifyPropertyChanged-liittymä on .NET-kirjastoon sisältyvä liittymä, joka sisältää tarvittavat toiminnot ilmoitusten (*Notifications*) lähettämiseen näkymän ja näkymämallin välillä [10]. Kun jokin näkymämallin ominaisuuden arvo muuttui, näkymän pitää saada tieto muutoksesta ja päivittyä sen mukaisesti. *IDisposable*-liittymä sisältyy niin ikään .NET-kirjastoon ja sisältää toiminnot näkymämallin poistamiseen [11].

Näkymämalleille, joita oli tarkoitus avata ja sulkea käyttöliittymässä, luotiin oma abstrakti työtilamalli WorkspaceViewModel. Työtilamalli periytyi perusmallista ViewModelBase. Työtilamalliin lisättiin toiminnot näkymän sulkemista sekä näkymämallien välisten viestien lähettämistä ja vastaanottamista varten. Työtilamallin pohjalta luotuja näkymämalleja hyödynnettiin käyttöliittymässä siirryttäessä näkymästä toiseen. Aina kun näkymää vaihdettiin, aikaisempi näkymä poistettiin ennen uuden näkymän avaamista.

Näkymämallien välisten viestien avulla työtilamallit pystyivät lähettämään viestejä muille näkymämalleille. Viestien avulla oli yksinkertaista toteuttaa käyttöliittymän odotusnäkymä. Kun työtilamalli aloitti tietojen lataamisen tai tallentamisen, se lähetti viestin ohjelman pääikkunalle. Työtilamalli lähetti viestin myös latausten ja tallennusten valmistumisesta. Näin pääikkuna pystyi käynnistämään odotusnäkymän latausten ja tallennusten ajaksi, eikä käyttöliittymä pysähtynyt mahdollisesti aikaa vievien toimintojen ajaksi.

Kuvassa 11 esitetään asiakasmallin näyttämiseen liittyvän CustomerViewModel-näkymämallin periytymistä. Kuvasta nähdään, että CustomerViewModel-näkymämalli periytyi suoraan abstraktista työtilamallista WorkspaceViewModel, joka periytyi edelleen abstraktista perusmallista ViewModelBase. Näin toteutettuna CustomerViewModel-näkymämalli peri ominaisuudet sekä työtilamallista että perusmallista.

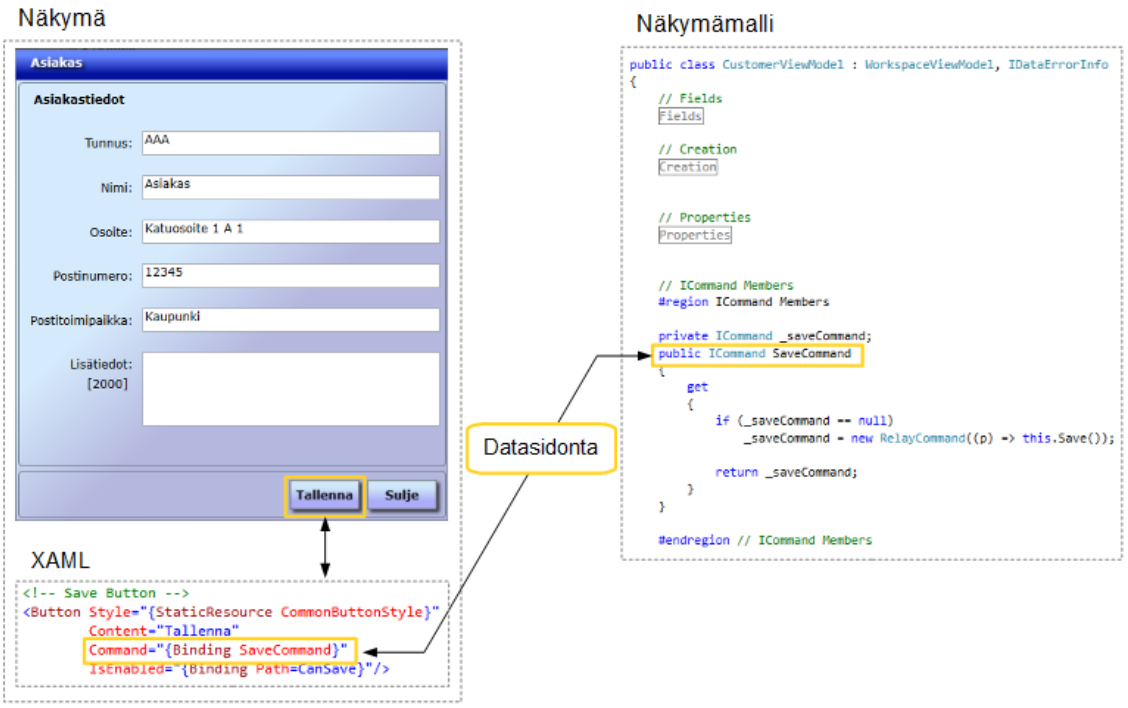


Kuva 11. CustomerViewModel-näkymämallin periytyminen

Abstraktien perus- ja työtilamalleista periytyvien liittymien ja ominaisuuksien lisäksi näkymämalleihin periytettiin myös *IDataErrorInfo*-liittymä. *IDataErrorInfo*-liittymän periyttäminen mahdollisti malliluokkien kelpoisuussääntötarkistusten hyödyntämisen näkymissä. Datasidonnan avulla tarkistusten tulokset voitiin sitoa suoraan näkymien elementteihin. Tämän ansiosta kelpoisuussääntötarkistuksia tarvitsi toteuttaa ainoastaan malleissa.

Käyttöliittymän näkymät (*View*) koostuivat kahdesta osasta: XAML-osasta sekä C#-osasta. XAML on Microsoftin kehittämä ja ylläpitämä XML-pohjainen kieli, jota käytetään käyttöliittymien graafisten elementtien kuvauskielenä [3]. C#-osa sisälsi ainoastaan näkymän luomiseen ja hävittämiseen liittyvät toiminnot. Kaikki muu toiminnallisuus oli toteutettu malleissa ja näkymämalleissa. XAML-osassa määriteltiin näkymän graafiset elementit, kuten tekstikentät, hyperlinkit ja painikkeet. Elementeille määriteltiin XAML-osassa niiden näyttämiseen liittyviä ominaisuuksia, kuten koko ja sijainti sekä mahdolliset tyylit, fontit jne. XAML-osassa määriteltiin myös elementtien datasidontaan käytetyt ominaisuudet eli mihin näkymämallin ominaisuuteen elementit oli sidottu.

Kuvassa 12 näytetään asiakastietojen muokkaamiseen kehitetyn näkymän datasidontaa näkymän ja näkymämallin välillä.

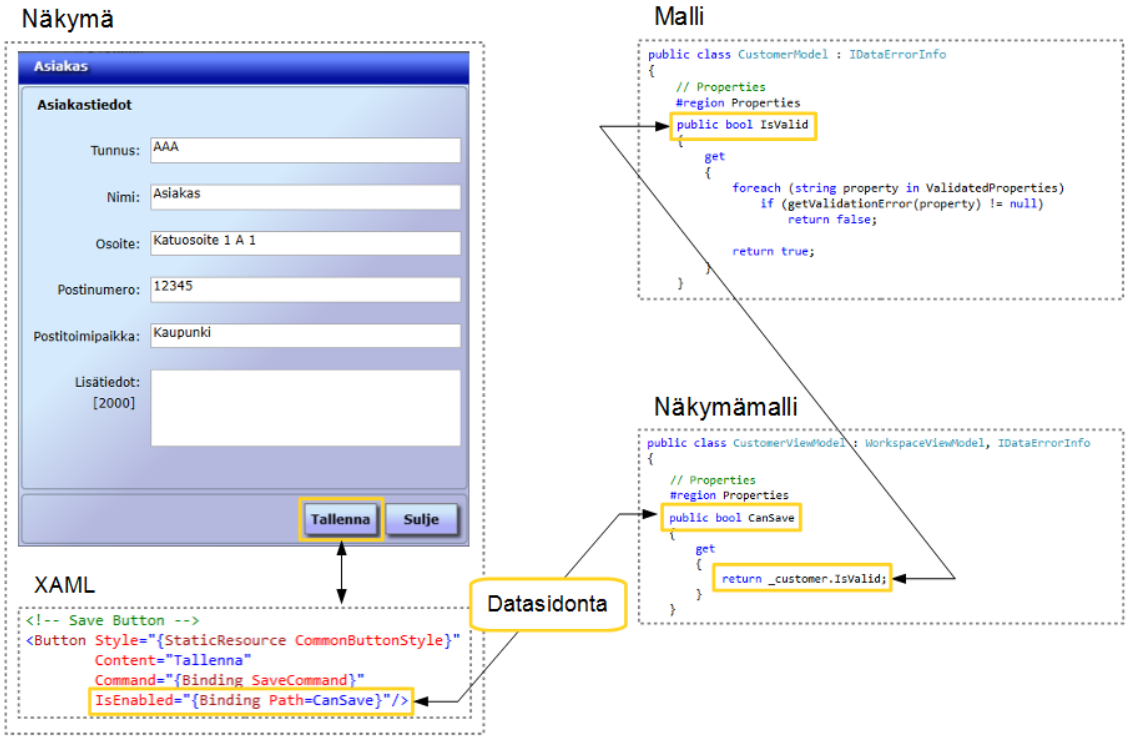


Kuva 12. Asiakastietonäkymän tallennuskomennon datasidonta

Kuten kuvasta 12 nähtiin, asiakastietonäkymän XAML-osassa määritetyn painikkeen varsinainen komento eli *Command*-ominaisuus oli sidottu datasidonnalla näkymämallin *SaveCommand*-ominaisuuteen. Datasidonnan ansiosta suoritus siirtyi näkymämallin *SaveCommand*-ominaisuuteen aina, kun näkymän Tallenna-painiketta painettiin. Vastaavasti tallennuspainikkeen aktiivisuutta kuvaava *IsEnabled*-ominaisuus oli sidottu näkymämallin *CanSave*-ominaisuuteen.

CanSave-ominaisuudessa hyödynnettiin mallin kelpoisuussääntöjä ja mallin kelpoisuutta kuvaavaa ominaisuutta *IsValid*. Näkymämallin *CanSave*-ominaisuuden arvo määräytyi mallin *IsValid*-ominaisuuden perusteella. Näin mallissa toteutettuja kelpoisuussääntöjä pystyttiin hyödyntämään komentojen yhteydessä. Kuvassa 13 näytetään asiakastietonäkymän tallennuskomennon aktiivisuuden määräävän ominaisuuden *IsEnabled* datasidonnan eteneminen mallin kelpoisuutta kuvaavaan ominaisuuteen *IsValid*.

Koska mallin ominaisuuteen *IsValid* ei sisältynyt mitään toimintoja näkymän päivittämiseen, näkymämallin tuli huolehtia päivitysilmoitusten lähettämisestä näkymälle. Näkymän päivitys toteutettiin suorittamalla *RaisePropertyChanged("CanSave")*-komento aina, kun tallennuspainikkeen aktiivisuus haluttiin tarkistaa ja näkymä päivittää. Käytännössä komento suoritettiin aina, kun minkä tahansa mallin *IsValid*-ominaisuuteen vaikuttavan ominaisuuden arvo muuttui.



Kuva 13. Datasidonnan eteneminen

Kuten kuvasta 13 nähtiin, tallennuspainikkeen *IsEnabled*-ominaisuus oli sidottu datasidonnalla näkymämallin *CanSave*-ominaisuuteen. *CanSave*-ominaisuuden arvo määräytyi puolestaan mallin *IsValid*-ominaisuuden perusteella. Tällä tavalla mallin ominaisuuksia ja niihin liittyviä kelpoisuussääntöjä voitiin hyödyntää näkymässä kuljettamalla ne näkymämallin kautta.

3.4 Raportointiominaisuuksien toteutus

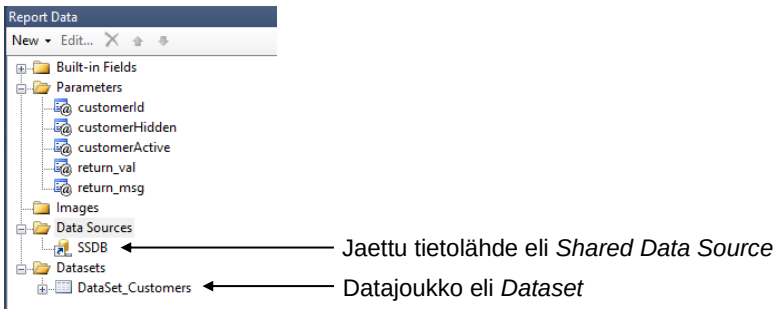
Tietojärjestelmän raportointitoiminnot toteutettiin hyödyntämällä SQL-palvelimen SSRS-palvelua. SSRS eli SQL Server Reporting Services on Microsoftin kehittämä ja ylläpitämä MS SQL-palvelinten lisäosa, joka mahdollistaa raporttien tallentamisen ja julkaisemisen SQL-palvelimella [2]. SSRS-palvelua hyödyntämällä järjestelmään sisällytetyt raportit voitiin toteuttaa MS Visual Studio -kehitysympäristössä ja julkaista ne SQL-palvelimella. SQL-palvelimella julkaistut raportit olivat käytettävissä myös selaimella, mikä paransi tietojärjestelmän käyttövarmuutta. Mikäli järjestelmään kehitetyn käyttöliittymän tai palvelun kanssa ilmenisi ongelmia, raportteja voitaisiin edelleen tarkastella ja esimerkiksi tulostaa suoraan SQL-palvelimelta.

Tietojärjestelmän vaatimusmäärittely- ja rajausvaiheessa järjestelmään sisällytettiin ainoastaan asiakaskohtainen kuukausiraportti. Tämän lisäksi järjestelmään päätettiin tehdä hakemistoraportti, jonka avulla asiakaskohtaiset kuukausiraportit olisivat helpommin löydettävissä myös selainta käytettäessä.

Raportit toteutettiin MS Visual Studio -kehitysympäristössä, jonka avulla raporttien asetuksia, tietolähteitä ja sivujen asettelua oli helppo muokata. Raporttien asettelussa huomioitiin se, että raportit oli tarkoitus tulostaa A4-kokoiselle paperille. Kaikki raporteissa näytettävien tietojen hakutoiminnot toteutettiin SQL-palvelimelle tallennetuilla proseduureilla. Proseduurien nimeämiskäytäntönä pidettiin sitä, että kaikki raportointiin liittyvät proseduurit nimettiin SSRS-alkuisiksi. Nimeämiskäytännöllä pyrittiin yksinkertaistamaan tallennettujen proseduurien tunnistamista.

Kuukausiraporteille luotiin yhteinen jaettu tietolähde eli *Shared Data Source*. Tietolähde määriteltiin siten, että se viittasi tietojärjestelmän päätietokantaan. Näin SQL-palvelimelle tallennetut funktiot ja proseduurit olivat raporttien saatavilla. Käyttämällä jaettua tietolähdettä varmistettiin, että eri raporttien tietolähteenä toimi aina sama tietokanta.

Raporteille luotiin jaettua tietolähdettä hyödyntämällä datajoukkoja eli *Datasets*. Raporteissa näytettävät yksittäiset tiedot saatiin datajoukoista. Kuvassa 14 (ks. seur. s.) esitetään raportissa käytettävä jaettu tietolähde ja datajoukko.



Kuva 14. SSRS-raportin jaettu tietolähde ja datajoukko

Jaetussa tietolähteessä oli määritelty tietokannan nimi ja SQL-palvelimen osoite. Siinä oli määritelty myös tietokannan käyttäjätunnus sekä salasana, joiden avulla tietolähde muodostaisi yhteyden SQL-palvelimeen. Datajoukoissa määriteltiin SQL-palvelimelle tallennettu proseduuri, mikä hakisi raportissa näytettävät tiedot tietokannasta. Tämän lisäksi datajoukossa määritettiin proseduurin tarvitsemat parametrit. Datajoukko voitiin määritellä myös siten, että tallennetun proseduurin sijaan määriteltiin suoritettava SQL-lauseke. Tässä tapauksessa SQL-lauseke hakisi raportissa näytettävät tiedot tietokannasta.

Kaikki tietojärjestelmän raporteissa käytetyt datajoukot toteutettiin siten, että datajoukossa määriteltiin SQL-palvelimelle tallennettu proseduuri ja sen tarvitsemat parametrit. Käyttämällä tallennettuja proseduureja suoritettavien SQL-lauseiden sijaan datajoukkojen tietoja voitiin muokata ja esimerkiksi järjestellä uudelleen muokkaamalla tallennettua proseduuria. Näin toteutettuna muutokset tulivat voimaan välittömästi, eikä itse raporttia tarvinnut muokata tai päivittää. Suoritettavien SQL-lausekkeiden muutoksissa raportti olisi pitänyt ensin päivittää ja julkaista sen jälkeen päivitetty raportti SQL-palvelimella.

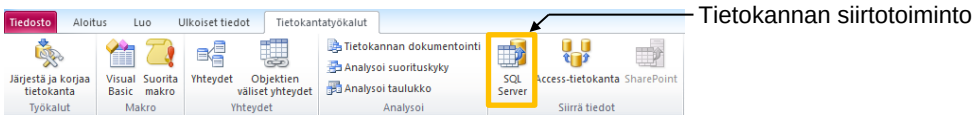
SQL-palvelimella julkaistut raportit näytettiin käyttöliittymässä hyödyntämällä .NET-kirjastoon sisältyvää *ReportViewer*-elementtiä. *ReportViewer*-elementti latasi ja näytti raportit sellaisenaan SQL-palvelimelta. Tämän ansiosta raporttien ulkonäkö ja toiminnot olivat samat kuin selainta käytettäessä. *ReportViewer*-elementtiin sisältyi myös toiminnot raporttien tulostamista ja tallentamista varten. *ReportViewer*-elementillä raportit voitiin tallentaa useassa eri muodossa, esimerkiksi Excel- tai PDF-muodossa. Kuvassa 15 (ks. seur. s.) esitetään *ReportViewer*-elementtiin sisältyviä toimintoja. Käyttöliittymään lisättiin *ReportViewer*-elementin lisäksi toiminto, millä oli mahdollista tulostaa useita raportteja kerralla. Tästä ominaisuudesta oli hyötyä etenkin kuukauden vaihtuessa, jolloin tulostettavia raportteja oli paljon.

Kuva 15. *ReportViewer*-elementin perustoiminnot

Raporttien julkaiseminen suoraan SQL-palvelimella mahdollisti raporttien muokkaamisen ja lisäämisen jälkeenpäin ilman, että koko järjestelmää tarvitsi päivittää. Tällä tavalla toteutettuna raportointiominaisuus, kuten koko tietojärjestelmä, pystyttiin pitämään mahdollisimman modulaarisena ja helposti muokattavana. Uusien raporttien lisäämisen yhteydessä oli huolehdittava ainoastaan siitä, että hakemistoraporttiin lisättiin linkki uuteen raporttiin. Näin lisätty raportti oli käytettävissä heti lisäyksen jälkeen sekä käyttöliittymällä että selaimella.

4 Tietojen tuominen MS Access -sovelluksesta

Tietojärjestelmään piti pystyä tuomaan vanhat tiedot aikaisemmin käytössä olleesta MS Access -sovelluksesta. Koska tiedot oli tarkoitus siirtää ainoastaan kerran, uuden järjestelmän käyttöönoton yhteydessä, tietojen siirtoon ei haluttu tehdä erillistä sovellusta. Tietojen siirtämiseen päätettiin käyttää MS Accessin Siirrä tiedot -toimintoa. Kuvassa 16 näytetään MS Accessin Siirrä tiedot -toiminto. Siirtotoiminnon avulla voitiin siirtää MS Accessiin tallennettu tietokanta sellaisenaan SQL-palvelimelle.



Kuva 16. MS Accessin tietojen siirto SQL-palvelimelle

Koska vanhan MS Access -tietokannan rakenne poikkesi huomattavasti työssä kehitettävän tietojärjestelmän päätietokannan rakenteesta, tuoduille tiedoille piti tehdä useita muunnoksia ja tarkistuksia ennen niiden tallentamista uuteen järjestelmään. Tarvittavat muunnokset ja muunnettujen tietojen tallentaminen päätettiin toteuttaa SQL-skripteillä. SQL-skriptit ovat SQL-ohjelmointikielellä toteutettuja toimintoja tai toimintasarjoja, joita voidaan suorittaa SQL-palvelimella. SQL-skripteillä voitiin tehdä hakuja vanhasta järjestelmästä tuoduille tiedoille ja suorittaa tarvittavia muunnoksia. Muunnosten jälkeen

tiedot voitiin tallentaa tietojärjestelmän päätietokantaan. Näin toteutettuna kaikki vanhan MS Access -sovelluksen tiedot saatiin siirrettyä uuteen järjestelmään.

Yksi vanhan sovelluksen isoista puutteista oli mahdollisuus tallentaa identtisiä asiakastietoja useampaan kertaan. Tämän takia osa tallennetuista töistä oli jakautunut useammalle asiakkaalle, jotka olivat todellisuudessa yksi ja sama asiakas. Siirron yhteydessä vanhoja tietoja tarkistettiin ja etsittiin esimerkiksi identtisiä nimiä asiakastiedoista. Identtiset tietueet yhdistettiin ja mahdolliset viittaukset muihin taulukoihin korjattiin. Näin uuteen järjestelmään tuli ainoastaan uniikkeja asiakastietoja ja tehdyt työt viittasivat oikeisiin asiakkaisiin.

Tuomalla tiedot vanhasta sovelluksesta hyödyntämällä SQL-skriptejä tietojen käsin syöttämisen sijaan vältyttiin mahdollisilta näppäilyvirheiltä ja tietojen tuominen onnistui huomattavasti nopeammin. Osalle tuotavista tiedoista piti tehdä niin suuria muutoksia tai vaativia tarkistuksia, että niiden toteuttaminen SQL-skripteillä ei ollut käytännössä mahdollista. Esimerkiksi laskutusosoitteiden tai puhelinnumeroiden paikkansapitävyys voitiin tarkistaa ainoastaan käsityönä. Tehtyjen töiden osalta tietojen tuominen ja tarkistukset pystyttiin toteuttamaan kokonaisuudessaan SQL-skripteillä.

Tietojen tuontiin kehitetyt SQL-skriptit oli toteutettu siten, että niissä hyödynnettiin tietojärjestelmän SQL-palvelimelle tallennettuja proseduureja. Näin tuotavat tiedot tallentuivat uuteen järjestelmään samalla tavalla kuin käyttöliittymän kautta tallennetut tiedot. Käyttämällä tallennettuja proseduureja kaikki oletusarvot ja tyyppimuunnokset toteutettiin samoin kuin järjestelmän normaalissa käytössä. Tällä varmistettiin se, että kaikki haku- ja raportointiominaisuudet toimisivat oikein myös vanhasta sovelluksesta tuoduille tiedoille.

5 Tietojärjestelmän käyttöönotto

Tietojärjestelmän varsinainen käyttöönotto suunniteltiin toteutettavaksi vasta kehitystyön valmistumisen jälkeen kesällä 2014. Kehitetylle tietojärjestelmälle tehtiin käyttöönottotestejä virtuaaliympäristössä. Virtuaaliympäristönä toimi VM VirtualBox. VM VirtualBox on Oraclen kehittämä virtuaalikoneympäristö, jonka avulla yhdessä tietokoneessa voidaan käyttää useampaa käyttöjärjestelmää samanaikaisesti [12]. VirtualBoxilla oli mahdollista luoda virtuaalikoneita, joihin oli asennettu tulevaa käyttöönottoympäristöä vastaava käyttöjärjestelmä.

Tällä tavalla voitiin testata tietojärjestelmän eri osien asentamista sekä niiden toimivuus ns. puhtaassa koneessa, mihin ei ollut asennettu ennestään mitään sovelluksia tai lisäosia. Tällä pyrittiin varmistamaan se, että kehitetty tietojärjestelmä ei riippunut millään tavalla kehitystyössä käytetyistä tietokoneista.

Virtuaaliympäristöllä voitiin testata myös järjestelmän toimivuutta hajautetulla kokoonpanolla. Näissä testeissä yhteen virtuaalikoneeseen asennettiin tietojärjestelmän käyttämä palvelu sekä SQL-palvelin ja toiseen virtuaalikoneeseen asennettiin ainoastaan käyttöliittymä. Näillä testeillä pyrittiin varmistamaan se, että järjestelmä voitaisiin tarvittaessa muuttaa useammassa tietokoneessa toimivaksi hajautetuksi järjestelmäksi.

Käyttöönottotesteissä virtuaalikoneissa suoritettiin samat toimenpiteen, mitkä tultaisiin tekemään myös varsinaisessa käyttöönotossa. Toimenpiteet olivat seuraavat:

- SQL-palvelimen ja siihen liittyvien päivitysten asennus.
- Vanhojen tietojen tuominen MS Access -sovelluksesta.
- WCF-palvelun asennus.
- WPF-käyttöliittymän asennus.

WCF-palvelulle ja WPF-käyttöliittymälle oli luotu kehitysvaiheessa asennusohjelmat helpottamaan niiden asennusta. WPF-käyttöliittymän asennuksessa hyödynnettiin ns. *ClickOnce*-menetelmää. *ClickOnce*-menetelmä on Microsoftin kehittämä ja ylläpitämä tekniikka, millä pystytään toteuttamaan sovellusten automaattiset päivitykset [13].

Käyttöliittymän osalta päätettiin käyttää *ClickOnce*-menetelmää, koska sen avulla oli yksinkertaista toteuttaa käyttöliittymän päivitykset. Tietojärjestelmää kehitettäessä arvioitiin, että suurin osa päivityksistä tulisi olemaan juuri käyttöliittymän korjauksia ja muutoksia. *ClickOnce*-menetelmää hyödyntämällä päivitykset voitiin toteuttaa järjestelmän kehitysympäristössä ja julkaista päivitetty versio verkon välityksellä.

Näin toteutettuna tietojärjestelmän käyttäjien ei tarvinnut itse tarkistaa ja asentaa päivityksiä, vaan päivitystarkistus tehtiin automaattisesti aina käyttöliittymän käynnistyksen yhteydessä. Vaikka tietojärjestelmä kehitettiin käytettäväksi ainoastaan yhdessä työasemassa, haluttiin usean käyttöliittymän tuki huomioida jo kehitysvaiheessa. *ClickOnce*-menetelmä mahdollisti myös useamman käyttöliittymän keskitetyn päivittämisen.

Näin käyttöliittymistä ei olisi käytössä eri versioita samanaikaisesti, vaan kaikki käyttöliittymät päivittyisivät *ClickOnce*-menetelmällä uusimpaan versioon.

WCF-palvelun ja SQL-palvelimen päivitykset ja korjaukset toteutettiin etätyöpöytäyhteydellä ja kopioimalla päivitetyt tiedostot tietokoneesta toiseen. Päivitysten tekeminen tiedostoja kopioimalla ei olisi järkevää käyttöliittymän yhteydessä, koska käyttöliittymiä saattaisi olla tulevaisuudessa useita. WCF-palvelun ja SQL-palvelimen päivittämiseen tämä tapa soveltui, koska tietojärjestelmässä oli ainoastaan yksi palvelu ja yksi SQL-palvelin.

6 Yhteenveto

Insinöörityössä saatiin kehitettyä tietojärjestelmä, joka täytti sille asetetut vaatimukset. Alkuperäisten vaatimusten lisäksi järjestelmään saatiin kehitettyä myös muutamia lisäominaisuuksia, joiden avulla järjestelmän käytettävyyttä saatiin parannettua. Näihin kuului esimerkiksi hakemistoraportti, joka kehitettiin helpottamaan kuukausiraporttien selaamista.

Järjestelmän käyttöliittymä saatiin kehitettyä toimivaksi kokonaisuudeksi, jonka ominaisuuksia on helppo muokata tulevaisuudessa. Käyttöliittymän lopullinen ulkoasu ja toiminnot tulevat tarkentumaan käyttöönoton jälkeen, kun järjestelmää käytetään sen varsinaisessa käyttötarkoituksessa. Työssä kehitetty käyttöliittymä toimii hyvänä pohjana tuleville muutoksille ja lisäyksille.

Järjestelmään tullaan todennäköisesti toteuttamaan myös uusia raportteja, kuten erityyppisiä yhteenvetoraportteja. Koska järjestelmän raportointiominaisuudet toteutettiin SSRS-raportteina, tulevien raporttien lisääminen ja jo olemassa olevien muokkaaminen on mahdollisimman yksinkertaista.

Kokonaisuudessaan työssä saatiin kehitettyä varsin toimiva ja helposti muokattava ja laajennettava tietojärjestelmä.

Lähteet

1 Haikala Ilkka. 2002. Ohjelmistotuotanto. Talentum Media Oy. Helsinki FI.

2 Features Supported by the Editions of SQL Server 2012 [WWW-dokumentti] http://msdn.microsoft.com/en-us/library/cc645993.aspx (Luettu 25.3.2014)

3 Introduction to WPF [WWW-dokumentti] http://msdn.microsoft.com/en-us/library/aa970268%28v=vs.100%29.aspx (Luettu 25.3.2014)

4 Moghadampour Ghodrat. 2011. C# Windows- ja tietokantaohjelmointi. WSOY-pro Oy, Docendo-tuotteet. Jyväskylä, FI.

5 Sovelto. 2013. SQL Server hallinta, seuranta ja optimointi [Kurssimateriaali]. FC Sovelto Oyj.

6 Löwy Juval. 2010. Programming WCF Services, Third Edition. O'Reilly Media Inc. Sebastopol, CA.

7 Allwork John. 2011. Visual Studio C# 2010 Programming and PC interfacing. Elektor International Media. UK

8 WPF Apps With The Model-View-ViewModel Design Pattern. [WWW-dokumentti] http://msdn.microsoft.com/en-us/magazine/dd419663.aspx (Luettu 25.3.2014).

9 5: Implementing the MVVM Pattern. [WWW-dokumentti] http://msdn.microsoft.com/en-us/library/gg405484%28v=pandp.40%29.aspx (Luettu 1.4.2014)

10 System.ComponentModel Namespace. [WWW-dokumentti] http://msdn.microsoft.com/en-us/library/vstudio/System.ComponentModel%28v=vs.100%29.aspx (Luettu 1.4.2014)

11 IDisposable Interface. [WWW-dokumentti] http://msdn.microsoft.com/en-us/library/system.idisposable.aspx (Luettu 1.4.2014)

12 Oracle VM VirtualBox. [WWW-dokumentti] https://www.virtualbox.org/ (Luettu 19.4.2014)

13 ClickOnce Deployment Overview. [WWW-dokumentti] http://msdn.microsoft.com/en-us/library/142dbbz4(v=vs.80).ASPX (Luettu 1.4.2014)